Bouraoui Hamadou

# Comparison of Equivalent Circuit Models with multiple Neural Networks

Bachelor Thesis

16 June 2023

Please cite as:
Bouraoui Hamadou, “Comparison of Equivalent Circuit Models with multiple Neural Networks,” Bachelor Thesis (Bachelorarbeit), Fachgebiet Elektrische Energiespeichertechnik, Technische Universität Berlin, Germany, June 2023.

Fachgebiet Elektrische Energiespeichertechnik
Technische Universität Berlin, Germany

Einsteinufer 11 · 10587 Berlin · Germany

https://www.tu.berlin/eet

# Comparison of Equivalent Circuit Models with multiple Neural Networks

Bachelor Thesis

vorgelegt von

**Bouraoui Hamadou**

geb. am 29. March 1996
in Sousse, Tunesien
Matrikelnummer: 395830

angefertigt am Fachgebiet

**Elektrische Energiespeichertechnik**

**Fakultät für Elektrotechnik und Informatik**
**Technische Universität Berlin**

| | |
|---|---|
| Betreuer: | **Dominic Kupfer** |
| Gutachterin: | **Prof. Dr.-Ing. Julia Kowal** |
| | **Prof. Dr.-Ing. Sibylle Dieckerhoff** |
| Abgabe der Arbeit: | **16. Juni 2023** |

# Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde.
Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

(Bouraoui Hamadou)
Berlin, den 16 June 2023

# Kurzfassung

Batteriemanagementsysteme sind für die Überwachung von Batteriezuständen unerlässlich. In der Tat, sie gewährleisten, dass die Batterien kritische Werte nicht überschreiten dürfen. Dies geschieht durch Implementierung von Modellen für Batterieüberwachungsaufgaben. Zu diesen Modellen zählen Ersatzschaltbild (ESB)-Modelle, physikalische Modelle und Machine Learning Modelle usw. . In dieser Arbeit werden zwei Modellkategorien untersucht: ESB-Modelle sowie auch neuronale Netze, die eine Unterkategorie von Machine Learning Modellen darstellen. Die Untersuchung befasst sich mit der Runtime Performance sowie auch Prädiktionsgenauigkeit im Bezug auf Batterieklemmspannung, wenn die beiden Modelltypen unter gleichen Bedingungen verglichen werden. Dabei wurde ein ESB der 2. Ordnung erstellt und damit Trainingsdaten generiert. Anschließend wurden zwei Modelltypen von neuronalen Netzen mit den von dem ESB-Modell generierten Daten trainiert und letztendlich auf die obengenannten Kriterien verglichen. Es hat sich dabei festgestellt, dass es auf dieser Weise möglich ist, gute neuronale Netze für Batterieüberwachungsaufgaben einzurichten, die eine ähnliche Genauigkeit wie die von dem ESB aufweisen, allerdings mit einer wesentlich schnelleren Laufzeit. Ferner wurde auch untersucht, ob es möglich ist, die Genauigkeit eines von den oben erwähnten neuronalen Netzen mit neuen nicht-ESB generierten Daten zu verbessern. Es hat sich ergeben, dass dies leider nicht möglich ist.

# Abstract

Batterie management systems are essential for battery state monitoring. They ensure that batteries do not exceed critical values and thus leading to, in some cases, hazards. To ensure this task, they rely on battery models for state estimation. Commonly used models are equivalent circuit models (ECM), physical models, and machine learning models. This thesis examines two categories of models: ECM models and neural network models, which are a subset of machine learning models. The two model types are going to be compared in terms of runtime performance and prediction accuracy in terms of terminal voltage estimation. For this purpose, a $2^{nd}$ order ECM was used to generate battery data. Subsequently, these data were used to train two neural networks. Finally, both ECM and neural network models were compared on the aforementioned criteria. It has been determined that it was possible to establish ECM-similar-performing neural networks for terminal voltage estimation, however, with significantly faster execution time. Furthermore, an examination was conducted to determine whether it is possible to improve the accuracy of the aforementioned neural networks using new non-ECM-generated data. It has been found that it is not the case.

# Contents

# Chapter 1

# Introduction

In the last few decades, battery management systems (BMS) have been steadily improved, allowing more resource-hungry models for battery monitoring tasks to be implemented and run efficiently. For these tasks, several empirical and physical grey box models have been elaborated. However, machine learning (ML) models in general and neural networks (NN) in particular, are state-of-the-art models that have become more prevalent in the context of batteries. On the one hand, ECM is more resource-friendly and highly valued for its ease of use and quick analysis. Nevertheless, they require specific data that needs the disassembly of the battery in question as well as lab measurements. On the other hand, NN can be easily adapted to new data and does not need specific experiments for parametrization. This implies that they could be a possible substitute for ECM.

To further explore this possibility, the purpose of this thesis is to compare the performance of common ECMs with various NN architectures under equal conditions for battery monitoring tasks such as state estimation. This comparison will additionally include the time and complexity required for the preparation and implementation of both model types. Furthermore, this thesis aims to explore the performance of NNs trained on ECM-generated data compared to the ECM in question, and how much non-ECM generated data does a NN model need increase its performance.

# Chapter 2

# Fundamentals

This chapter explains the theoretical background needed to understand this thesis. Section 2.1 presents the core idea of the ECM in battery state estimation. Section 2.2 introduces the basic concept of NN and presents two different NN architectures that are going to be examined in this thesis. Finally, section 2.3 mentions the used hardware for this thesis and highlights some advantages of PyTorch for developing NN. I would like to highlight that the physical models, even if mentioned in the introduction, won't be discussed in this thesis. However, I would like to point to the Literature pages *[1, Pages 65-135]* where these model types were discussed in detail:

## 2.1 ECM

*According to Plett [1, page 29], an ECM is a simple approach to modeling a battery cell operation by using analog electrical components to define a behavioral approximation of a cell's voltage response to different input-current stimuli.* Figure 2.1 represents an example of a 2$^{nd}$ order ECM. If we were to open the cell, obviously, we won't be finding these components.

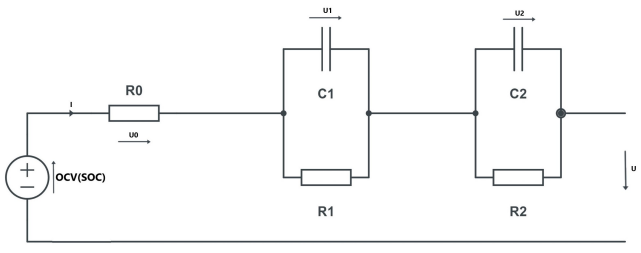


**Figure 2.1** – 2$^{nd}$ order ECM inspired from [2, page 3]

As we can see, the 2nd order ECM, element by element, is made out of:

1. **Open Circuit Voltage (OCV):** models the voltage drop across the terminals of a cell depending on its state of charge (SOC). The OCV of a fully charged battery will defer from the OCV of the same fully discharged battery [1, page 30].

2. **Intern resistance** $R_0$: models the dynamic behavior of a battery subjected to a time-variant input current [1, page 33]. When subjected to a load, the cell's terminal voltage drops below its OCV and rises above the OCV when it is being charged.

3. **2 RC-circuits in series:** These elements model the polarisation effect of the battery cell. *In agreement to Plett [1, page 34] "Polarization refers to any departure of the cell's terminal voltage away from the OCV due to a passage of current through the cell".* If we were to develop an ECM with only the two first-mentioned elements, the polarization would happen instantaneously following the Ohm-Law $i(t) \cdot R_0$. However, real battery cells behave more complexly.

   *Plett [1, page 34] offers a good example of this complex behavior "...when the battery of a flashlight is just about to be empty, its emitted light slowly starts to grow dimmer until it turns off completely. Turning the flashlight off, waiting for some time, and then turning it on makes it work again. ...".* This phenomenon is due to a process called "diffusion voltage" caused by a slow diffusion process in a lithium-ion cell, which can be approximated by using one or more RC-circuits in series *[1, page 35]*

Inspired from the work of [3, pages 3-5], the equations 2.1 and 2.2 can be derived as following:

Using Kirchoff's law and the mesh equation for the 1st RC-circuit, we have:

$$\begin{aligned} I(t) &= \frac{U_1}{R_1} + C_1 \cdot \frac{dU_1}{dt} \\ \iff \frac{dU_1}{dt} &= \frac{I(t)}{C_1} - \frac{U_1}{R_1 \cdot C_1} \end{aligned}$$

Similarly, we get for the 2nd RC-circuit:

$$\frac{dU_2}{dt} = \frac{I(t)}{C_2} - \frac{U_2}{R_2 \cdot C_2}$$

To express the change in SOC value:

$$\begin{aligned} SOC(t) &= SOC(t = t_0) + \frac{1}{C_{\text{Ref}}} \cdot \int_{t_0}^{t+t_0} I(t)\, dt \\ \iff \frac{dSOC}{dt} &= \frac{1}{C_{\text{Ref}}} \cdot I(t) \end{aligned}$$

Considering now the mesh of the entire ECM, we get for the terminal voltage expression:

$$U = OCV(SOC) - U_1 - U_2 - I(t) \cdot R_0$$

The matrix representation of these equations is displayed in equations 2.2 for the terminal voltage and 2.1 for the state vector of the ECM

$$\begin{aligned} \begin{bmatrix} \dot{SOC} \\ \dot{U_1} \\ \dot{U_2} \end{bmatrix} &= \begin{bmatrix} 0 & 0 & 0 \\ 0 & -\frac{1}{R_1 \cdot C_1} & 0 \\ 0 & 0 & -\frac{1}{R_2 \cdot C_2} \end{bmatrix} \cdot \begin{bmatrix} SOC \\ U_1 \\ U_2 \end{bmatrix} + \begin{bmatrix} \frac{1}{C_{Ref}} \\ \frac{1}{C_1} \\ \frac{1}{C_2} \end{bmatrix} I \\ &= \begin{bmatrix} 0 & 0 & 0 \\ 0 & -\frac{1}{\tau_1} & 0 \\ 0 & 0 & -\frac{1}{\tau_2} \end{bmatrix} \cdot \begin{bmatrix} SOC \\ U_1 \\ U_2 \end{bmatrix} + \begin{bmatrix} \frac{1}{C_{Ref}} \\ \frac{1}{\frac{\tau_1}{R_1}} \\ \frac{1}{\frac{\tau_2}{R_2}} \end{bmatrix} I \end{aligned} \tag{2.1}$$

$$\begin{bmatrix} U \end{bmatrix} = \begin{bmatrix} OCV(SOC) & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ U_1 \\ U_2 \end{bmatrix} + \begin{bmatrix} R_0 \end{bmatrix} \cdot I \tag{2.2}$$

## 2.2 Neural Network

*Stevens, Antiga, and Viehmann [4, page 3-5] defined artificial intelligence (AI) as a set of algorithms that aim to develop computer systems able to approximate complicated, nonlinear processes effectively* ***only through examples****, which we can use for performing tasks automatically that were previously limited to humans. Machine learning (ML) is a subfield of AI that relies heavily on* ***feature engineering****, which consists of finding and crafting the proper transformation of the data so that the algorithm can solve the task at hand. The real challenge here is discovering these transformations, which are not always evident initially. Deep learning (DL), however, is a subset of ML that deals with finding these transformations automatically from raw data in order to complete the task. The drawback is that it usually needs large amounts of data to approximate these complex processes.*

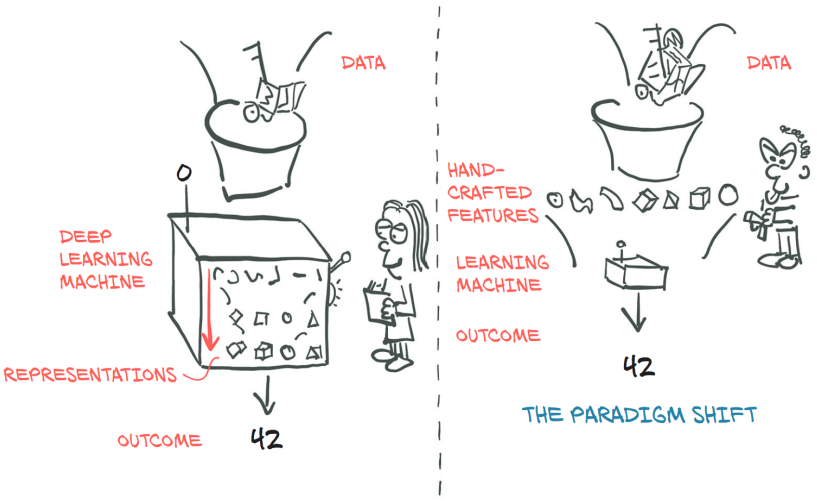


**Figure 2.2** – Grafical representation of the difference between DL (on the left) and ML (on the right). The features are being "manually" handcrafted on the right side, whereas on the left side, they are found automatically [4, page 5]

*To execute this massive challenge, deep learning relies on computational models called neural networks (NN), inspired by the human brain. NN consist of artificial neurons organized in layers; as for example the feed-forward NN (FFNN) in figure 2.3.*

*Patterson and Gibson [5, page 104-106] defined in their book that a FFNN typically has one input layer, one output layer, and in between one or more hidden layers. Each neuron of a hidden layer is usually connected with all neurons of the previous layer and all neurons of the next layer. Each layer processes information in a certain way (which will be explained in section 2.2.1) and passes it to the next layer, allowing the network to learn patterns from the raw input data and then make predictions.*

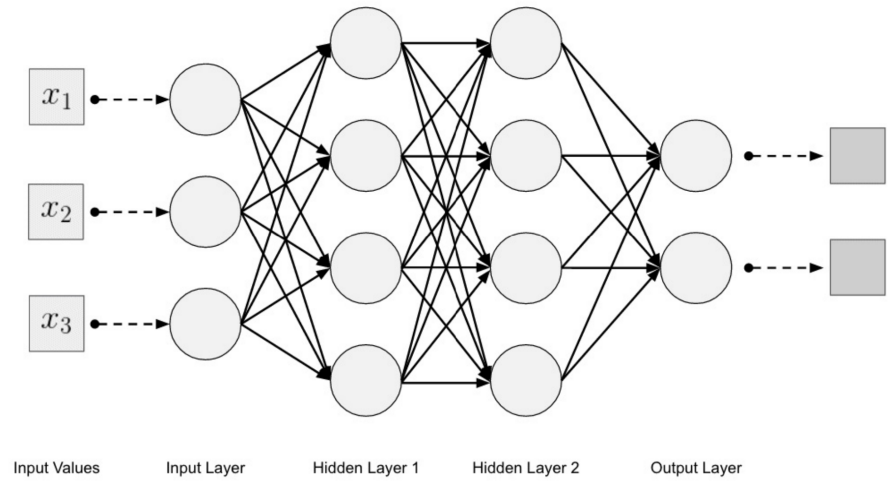


**Figure 2.3** – Feed-forward NN topology *[5, page 79]*

I want to bring to attention that NNs have seen a lot of progress in terms of their architecture since their creation *[5, page 164]*. Explaining the math behind the ability of the NNs and their architectures to process data, extract features, make predictions, train, and learn from errors would go beyond the scope of this thesis. The goal is to enable the readers to gain some background knowledge to understand this work and not to delve into the NN concept. Since the FFNN is the simplest NN architecture, it is going to be the backbone of this section as well as of the following subsections:

- **Mechanics of learning**
- **Convolutional Neural Network - CNN**
- **Long Short Term Memory - LSTM**

### 2.2.1 Mechanics of learning

*Stevens, Antiga, and Viehmann [4, page 104-105] introduced this concept by starting with a straightforward example of Kepler's first and second laws. Kepler* ***tried different shapes, utilizing some of the available measured data, then employed these shapes to determine the position of the planets based on the remaining measured data****. The authors finished by summarizing Kepler's train of thought as follows:*

1. *Gather the data.*
2. *Visualize the data.*
3. *Choose the simplest possible model to fit the data (in the case of Kepler's law an ellipse).*

4. *Split the data into two sets: a set to work with and another to validate.*
5. *Start with some parameters for the ellipse (eccentricity and size) and iterate until the model fits the observations.*
6. *Validate the model on the remaining data.*

The training of the NNs are based on the above steps but with some changes:

- First, we gather the data and preprocess it. NNs typically expect that the input data needs to be scaled down between [0,1] or [-1,1] [5, page 74]. They need to be stored in a special kind of "container" called a **tensor** that we will describe later in subsection 2.3.
- Second, we plot the available data, in the case that we are not familiar with it.
- Third, we choose the right NN architecture, the adequate activation function, and the right loss function, depending on the type of data and the task at hand.
  - FFNN is a general type of NN that can be used to complete various tasks and thus handles different types of input: image regression, image classification, object recognition, time series, etc., it is unfortunately not the most suitable NN architecture for specific problems, since it has certain limitations [6, page 8]. For instance, FFNNs lack memory and feedback connections, which makes them less effective in capturing temporal dependencies in sequential data (see section 2.2.3). Another limitation would be the increase in learning parameters and, therefore, an increase in computational power and time when training on grid-like data such as images (see section 2.2.2).
  - To understand the concept of information procession by each layer, we have to explore how an individual neuron processes data and how this information is propagated through the network layers.

    As mentioned by Stevens, Antiga, and Viehmann [4, pages 142-143] artificial neurons make the building block of how these layers process information. At its core, a neuron takes in inputs, performs computations on them by multiplying the input with a number called "weight" and adding a constant called "bias", and finishes by combining both of them using a typically fixed nonlinear function referred to as the "activation function". The following figure 2.4, represents graphically the mathematical operation of a neuron described above.

    The layers are composed of a parallel stack of neurons. The output of each layer in a FFNN is forwarded to all neurons of the next layer. This means that each neuron of the next layer takes the input of the entire

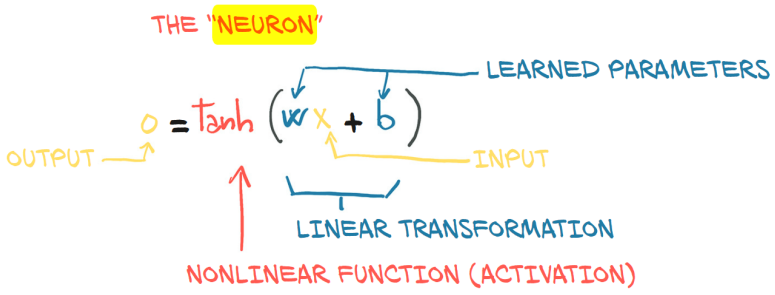


**Figure 2.4** – Mathematical operation of a neuron[4, page 143]

previous layer. A FFNN as represented in figure 2.3, is made up of a combination of these basic functions represented in figure 2.4. Figure 2.5 describes the mathematical representation of an output of the FFNN shown in figure 2.3.

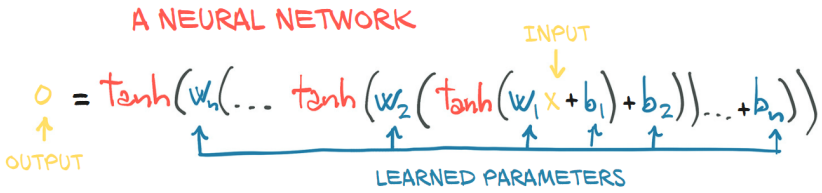


**Figure 2.5** – An output of a FFNN is made of a composition of functions represented in figure 2.4[4, page 144]

More about activation functions can be found in Pedamonti [7]. Pedamonti [7] provides a comprehensive comparison of different activation functions. *Stevens, Antiga, and Viehmann [4, page 148] discuss some generalities about activation functions, that if not present, the NN falls back to being a linear model or becomes very difficult to train*

- ***The activation function must be nonlinear to approximate nonlinearity***.
- ***The activation function must be differentiable to compute the gradient and minimize the error***.

– We must use the right loss function, on top of the right architecture and the adequate activation function, in order to calculate the gradient and minimize error. *Stevens, Antiga, and Viehmann [4, page 109] defined a loss function "as a function that computes a single numerical value that the learning process will attempt to minimize. The calculation of loss typically*

*involves the difference between the desired outputs for some training samples and the outputs actually produced by the model when fed those samples*". The choice of the loss function depends on the nature of the task at hand. In this thesis, we are dealing with a regression task. Chollet [8, page 114] suggest for this case the use of the "mean square error" (MSE) loss function. Other loss functions and their use can be found on the same page.

- Fourth, we split the data into training and validation sets according to figure 2.6.

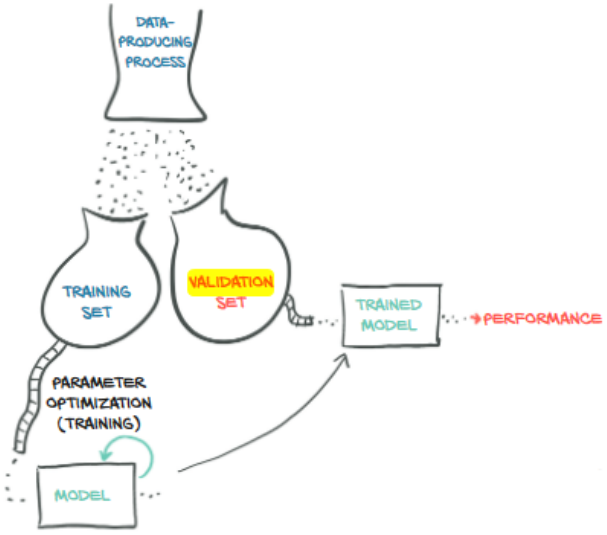


**Figure 2.6** – Split data in training and validation sets [4, page 132]

- The fifth and sixth steps consist of a training loop. Figure 2.7 summarizes the training process of a FFNN.

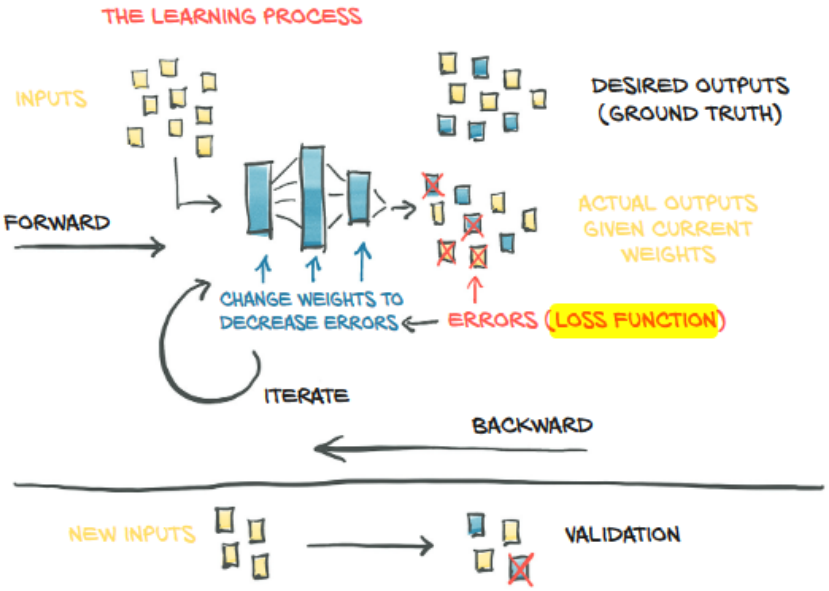


**Figure 2.7** – Learning process of a FFNN [4, page 142]

– One training loop is called an "epoch" [4, page 116]. The model's training generally consists of multiple epochs [4, page 135]. Each epoch consists of loading the input data in the input layer of the FFNN. It gets propagated to the next layers, where it gets processed and transformed, as explained previously. The process is repeated until reaching the output layer. The loss gets computed with the chosen loss function by passing the generated output of the FFNN and the desired output as arguments to the function. As explained by Stevens, Antiga, and Viehmann [4, page 106], the parameters (the weights and biases) of the model get optimized by calculating the gradient of the error with respect to these parameters, then updated in the direction that leads to a decrease in error. This operation is called backpropagation [4, page 123]. The backpropagation happens only on the training set and not on the validation set. As seen in figure 2.7, the validation set is used to check if the model performs well on unseen data [4, page 132].

### 2.2.2 Convolutional Neural Network - CNN

Stevens, Antiga, and Viehmann [4, pages 174, 189, 190, 194, 195] addressed the problem of using a FFNN in an image recognition task: differentiating airplanes from birds. On the one hand, it led to using a tremendous amount of parameters, resulting in more training time and increased computer resources. On the other hand, the

model performed poorly on unseen data where the object to recognize is moved, as no position independence had been reached. To overcome these issues, the model had to see all possible positions of the object in question to force generalization. But this solution still didn't address the issue of having too many parameters. The authors transitioned to the idea of using the neighboring pixels (instead of all pixels when using the FFNN) and thus to the solution of using discrete convolutions. *Stevens, Antiga, and Viehmann [4] defined a convolution on a 2D image as the scalar product of a weight matrix (the kernel), with every neighborhood in the input*.

In other words, the kernel is transitioned over the image, and the output is calculated in each transition as in figure 2.8. Graphically, after positioning a kernel with a 3x3 size on an image with a 4x4 size, each kernel element gets multiplied by its corresponding pixel of the image. The results of these multiplications are then summed, and the kernel is transitioned to another position.

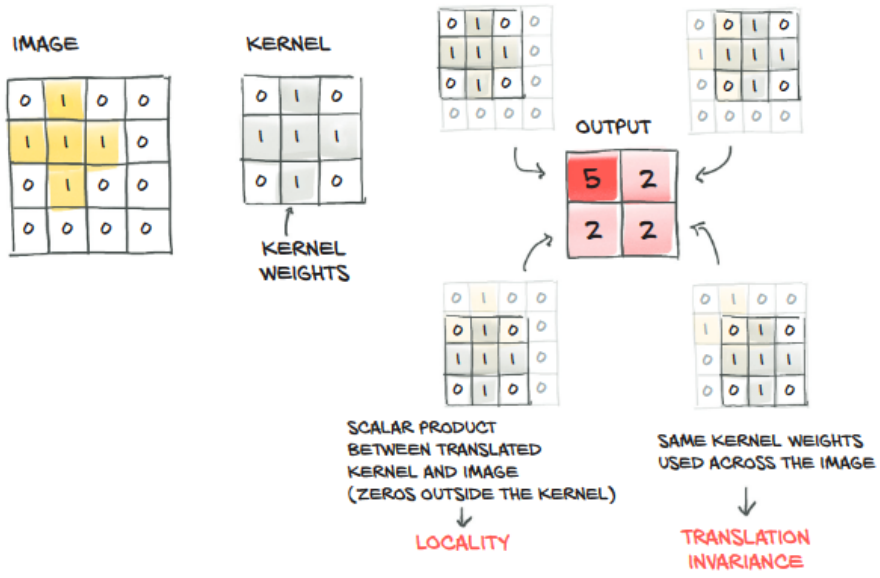


**Figure 2.8** – Grafical representation of a convolution with a kernel size of 3x3 and an image size 4x4 [4, page 196]

PyTorch offers convolutions for one, two, and three dimensions [4, page 196]:

- **nn.Conv1d** for time series
- **nn.Conv2d** for images
- **nn.Conv3d** for videos or volumes

In chapter 3, we will deal with time series. In this case, the demonstrated convolution operation is as follows:

Input Sequence: $[0, 1, 2, 0]$

Kernel Weights: $[0.5, 1, 0.5]$

First Convolution:

$(0 \times 0.5) + (1 \times 1) + (2 \times 0.5) = 2.0$

Second Convolution:

$(1 \times 0.5) + (2 \times 1) + (0 \times 0.5) = 2.5$

### 2.2.3 Long Short Term Memory - LSTM

Patterson and Gibson [5, pages 283-290] described the LSTM as being a variation of the recurrent NN (RNN), the latter being a superset of FFNN but with the concept of recurrent connections as shown in figure 2.9. At each time-step of feeding the network with input data, data is also being fed to the NN from the hidden states. The reason for the development of the LSTM was the issue with RNN's vanishing gradient. This happens when the gradient is too big or too small and makes it difficult to model long-range dependencies (10 time steps or more) in the structure of the input dataset.

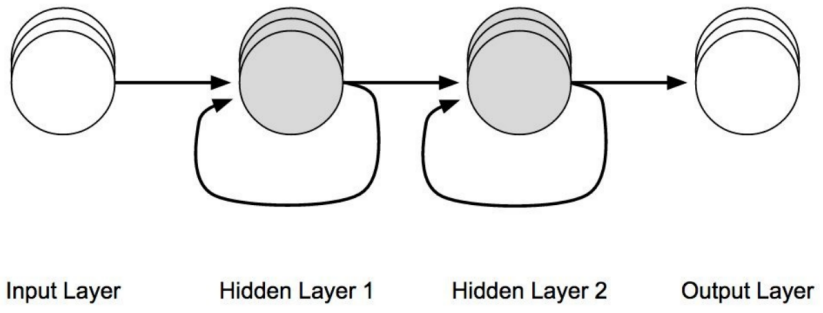


**Figure 2.9** – Architecture of RNN: recurrent connections on hidden-layer nodes [5, page 287]

More explanations on how the LSTM works, are not required to understand the work in section 3. However, I strongly advise the readers of my thesis to read the following literature by Patterson and Gibson [5, pages 274-303].

## 2.3 Used hardware and library

Tabel 2.1 represents the used hardware to write the necessary code in chapter 2.3

**Table 2.1** – Used hardware: desktop setting

| Hardware | Characteristics |
|---|---|
| CPU | Ryzen 5 3600: 6 cores/12 threads 3,6GHz up to 4.2GHz |
| RAM | 64GB DDR4 @2133MHz |
| GPU | Inno3d RTX 3060 Twin X2 12GB |
| Motherboard | MSI A320M-A PRO Max |
| Power supply | CSL 500W with 82% efficiency |
| Hard drive | Toshiba M.2 SSD RD500<br>Read speed: 3400 MB/s<br>Write speed: 2500 MB/s |
| Operating system (OS) | Windows 10 Pro |

The PyTorch library was utilized to construct the NN architectures discussed in subsections 2.2.2 and 2.2.3. In addition to its application in this thesis, PyTorch offers a lot of advantages when developing NN models [4, pages 7-14, 39–69]:

- it offers high-performance code since, at its core, it is written in C++ and CUDA.
- it allows the transfer of the computations from the Central Processing Unit (CPU) to the Graphics Processing Unit (GPU) with a function call. Moving the computations to the GPU means a decrease in computation time, usually 40 to 50 times, compared to the equivalent computation time on the CPU.
- it uses the Python Application Programming Interface (API) that makes it easy to use for researchers who have used Python previously.
- its core modules for building NN are located in "torch.nn" as examples: NN layers, loss functions, activation functions, etc.
- it enables the user to deploy their models on mobile devices ([4, chapter 15] for more information)
- it presents an abstract way of storing data in custom datasets and iterating over the datasets with a "DataLoader" ([4, chapter 7] for more information)

- it provides the users with an easy-to-use tensor, a data structure capable of storing data represented in figure 2.10. *As defined by Stevens, Antiga, and Viehmann [4, page 41], in the context of deep learning, "tensors refer to the generalization of vectors and matrices to an arbitrary number of dimensions*". It is worth noting that the inputs and outputs of NN architectures in PyTorch are expected to be tensors.

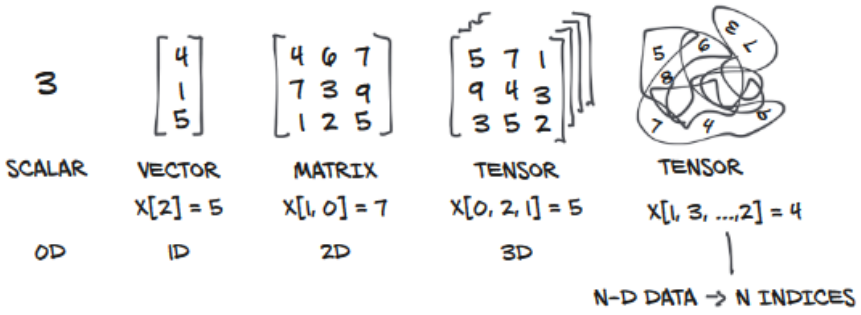


**Figure 2.10** – Graphical representation of a tensor [4, page 41]

# Chapter 3

# Developed architecture

This chapter is based on the fundamentals in chapter 2 and describes the code development steps that allow us to compare the developed models further explained in chapter 4. It is worth bringing up that **all developed models were limited to the terminal voltage prediction**. In this work, I am assuming that the reader is familiar with some programming concepts, for example, object-orientated programming, libraries, for example, numpy and pandas, as well as some Python data structures, like dictionaries and lists. If this is not the case, I strongly advise the readers to consult the available online documentation. This is especially true when using the PyTorch and Optuna libraries in sections 3.3 and 3.4. Throughout this chapter, the focus will be on discussing the objectives of the functions. The implementation details, including the provided parameters and return values, can be found in the comprehensive docstring accompanying the code. This chapter addresses the following key points:

1. introducing the given data
2. implementing an $n^{th}$ order ECM class
3. generating NN training data with the ECM
4. implementing the NN architectures in 2.2.2 and 2.2.3, as well as training them on the ECM generated data
5. implementing an evaluation code for the above-defined models

Most of the used functions and classes were defined in the Python file **tools.py**. Please consult the respective docstring file, if there are any ambiguities with the following defined functions.

## 3.1 Given data

Table 3.1 represents the data file names used throughout this thesis, which are essential for the simulation and evaluation of the experiments in the next chapter 4.

| File name | Representation | Source |
|---|---|---|
| LGHE4C25B01 | Matlab data file storing some battery measurements: U, I, capacity, total capacity during charge and discharge at 15, 25, and 45°C. Used to evaluate the developed models | Given by the supervisor |
| LGHE4C25B01_Parameter | Matlab data file storing the 2nd-order-ECM parameters defined in the theory part | Given by the supervisor |
| BMW i3 60W dataset | Dataset used when a NN model needs to be retrained | Downloaded from |
| FUDS dataset | Federal Urban Driving Schedule: dataset used to evaluate the developed models | Given by the supervisor |

**Table 3.1** – Used data files and their representation

The extraction of the Matlab files occurred with the library **mat4py** and happens respectively with **load_data_LGHE4C25B01()** and **load_data_LGHE4C25B01_Parameter()**, where:

- load_data_LGHE4C25B01() returns in this order a charge dictionary and a discharge dictionary for the measured battery data. The first key represents the temperature (either 15, 25 or 45°C) as a string. The second key represents the logged data.
- load_data_LGHE4C25B01_Parameter() returns in this order a charge dictionary, a discharge dictionary and a mean dictionary for the ECM parameters. The first key represents the temperature (either 15, 25 or 45°C) as a string. The second key the parameter themselves.
- The BMW i3 data consists of 72 real driving trips with a BMW i3 (60 A h). Each of those trips is stored in .csv dataset. Each dataset had to be scaled down to match the battery specifications used in this thesis. This has been

done by calculating $C_{\text{series}}$ and $C_{\text{parallel}}$ as follows [1, page 4]:

$$C_{\text{series}} = \frac{max(U_{\text{i3}_{\text{max}}})}{\text{eoc}} \tag{3.1}$$

$$C_{\text{parallel}} = \frac{60\,\text{A h}}{C_{\text{ref}}} \tag{3.2}$$

$U_{\text{i3}_{\text{max}}}$ is dependent on the driving dataset in question, whereas the end of charge (eoc) and the total capacity of the battery ($C_{\text{ref}}$) on the given measurements of the battery (see table 3.1).

- The FUDS dataset was already scaled down to the battery specifications. No processing in the next subsections happened on it.

## 3.2 ECM implementation

The ECM implementation is a straightforward implementation of the theory in subsection 2.1; however, extended to the implementation of an $n^{\text{th}}$-order-ECM. There is a slight difference in implementing the ECM class to calculate the state vector. The first element of the square matrix was set to a value 1, subsequently resulting in **the SOC value and the not** $\dot{SOC}$ as the first element of the state vector. This was done purposely to update the OCV value of the ECM without requiring additional code.

$$\begin{bmatrix} SOC \\ \dot{U}_1 \\ \dot{U}_2 \\ \vdots \\ \dot{U}_n \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & -\frac{1}{\tau_1} & 0 & \cdots & 0 \\ 0 & 0 & -\frac{1}{\tau_2} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & -\frac{1}{\tau_n} \end{bmatrix} \cdot \begin{bmatrix} SOC \\ U_1 \\ U_2 \\ \vdots \\ U_n \end{bmatrix} + \begin{bmatrix} \frac{1}{C_{Ref}} \\ \frac{1}{\frac{\tau_1}{R_1}} \\ \frac{1}{\frac{\tau_2}{R_2}} \\ \vdots \\ \frac{1}{\frac{\tau_n}{R_n}} \end{bmatrix} \cdot I \tag{3.3}$$

$U_1, U_2, \ldots, U_{\text{n}}$ represent the voltage accross the RC-circuits, $\tau_1, \tau_2, \ldots, \tau_{\text{n}}$ represent their respective time constants.

The terminal voltage of the ECM would be then:

$$\begin{bmatrix} U \end{bmatrix} = \begin{bmatrix} OCV(SOC) & 1 & 1 & \cdots & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ U_1 \\ U_2 \\ \vdots \\ U_{\text{n}} \end{bmatrix} + \begin{bmatrix} R_0 \end{bmatrix} \cdot I \tag{3.4}$$

It is again worth mentioning that the extraction of the ECM parameters defined in equations 3.3 and 3.4 is based on the charge and discharge measurements of the battery. This won't be explained further in this thesis. For readers who are wondering how parameter fitting occurs, I would like to point them to the following papers: Hu, Li, and Peng [9], Zhang, Zhang, and Lei [10], Jiang [11], as well as Ruba et al. [3].

The reason for developing an $n^{th}$-order ECM was for code reusability. In chapter 4, a $1^{st}$-order ECM is compared with a $2^{nd}$-order ECM. One should absolutely refrain from creating two classes that behave in a similar way when coding. Such a practice is deemed unacceptable as it can lead to unmanageable code, especially when the project's complexity and size increase.

For this purpose, 3 functions have been written by myself to initialize and simulate the ECM:

- **load_data_LGHE4C25B01_Parameter()** returns the given ECM parameters at a specific temperature (either 15°C, 25°C or 45°C).
- **interpolate_ecm_parameter()** takes the loaded ECM parameters from the previous function as input and returns a dictionary containing interpolations of the ECM parameters as represented in figure

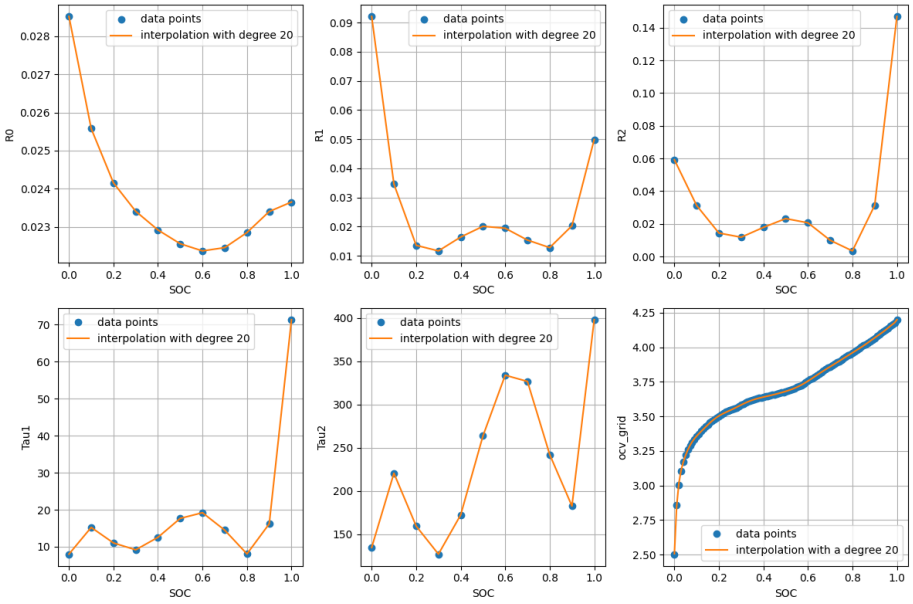


**Figure 3.1** – Example of ECM parameter interpolation. Mean parameters were interpolated with a polynom of the $20^{th}$ degree.

- **simulate_ecm()** simulates a given ECM with a specific passed current array. The function iterates over the array and updates the ECM's state vector follow-

ing the equation 2.1. Once the iteration is complete, it returns a dictionary containing the record of the ECM's behavior through the simulation.

During the testing of the ECM, I encountered an important error related to the unit of the capacity of the battery (default A h and of the time step (default s). **It is crucial that both variables have the same unit. Otherwise, the ECM's behavior becomes erroneous.** This should be revised in future improvements of my work.

Both 1$^{st}$ and 2$^{nd}$ order ECMs were tested and compared against each other (see next chapter 4) and the best model was used for the generation of the training data for the NNs in sections 3.3 and 3.4. The ECM training data generation consisted of 3 battery schedules that where defined as function to improve the code readability:

- **Full cycle (FC):** its function is full_cycle_data.
- **Pulse cycle (PC):** its function is pulse_cycle_data
- **incremental Open circuit voltage (iOCV):** its function is incremental_ocv_-data

All these schedules were stored in a nested dictionary named "schedules" that was passed to the above-mentioned functions as an argument.

The first key of the dictionary "schedules" was a string containing one of the above abbreviations. Each value was another dictionary containing the nature of the schedule to be run on the ECM. In all three schedules, a current list was defined with values 0.5 A, 1 A and 2 A. During charge, the ECM was subjected to a constant positive current, while during discharge, the current was negative (but still constant).

The differences between the three schedules were the other values besides the current.

**FC** consisted of a schedule where the ECM, starting from a SOC of 0.01, was fully charged and later on completely discharged with a constant current. Figures 3.2 and 3.3 represent graphically a data sample of the terminal voltage as well as of the current from the FC schedule.

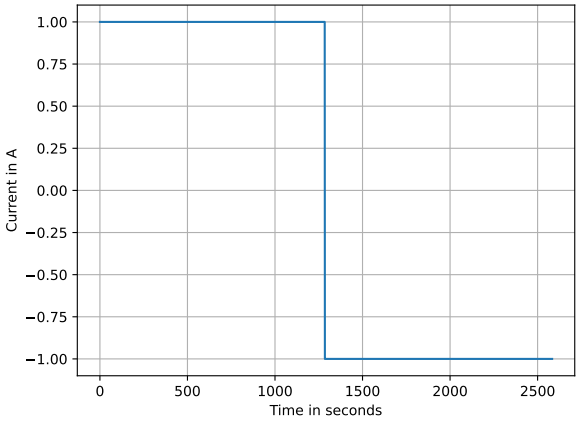


**Figure 3.2** – Current sample from the FC schedule at 15°C.

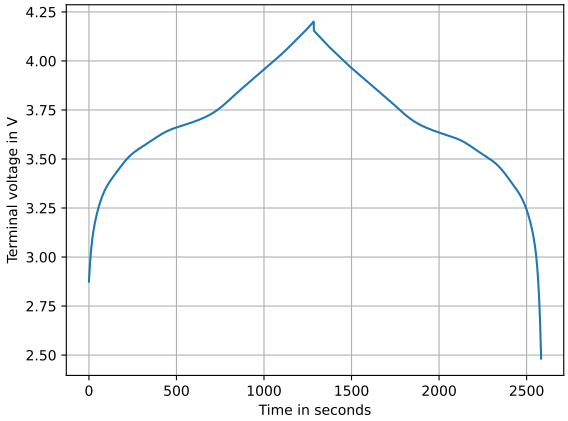


**Figure 3.3** – Terminal voltage sample from the FC schedule at 15°C.

**PC** consisted of subroutines where the ECM was subjected to two pulse cycles: the first one for charging and the second for discharging. The length of these cycles was defined by the depth of discharge (DOD) and the SOC. When the ECM is being instantiated, its SOC got set to $mSOC - \frac{DOD}{2}$. "mSOC" and "DOD" were both two different lists containing SOC and DOD values. The ECM gets charged until it reaches a SOC value of $mSOC + \frac{DOD}{2}$ then discharged back to the initial one $mSOC - \frac{DOD}{2}$. Figures 3.4 and 3.5.

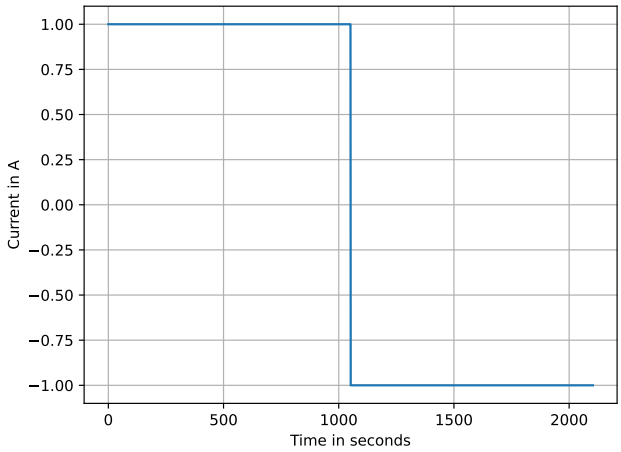


**Figure 3.4** – Current sample from the PC schedule at 15°C, with I=1 A, mSOC=0.4 and DOD=0.8.

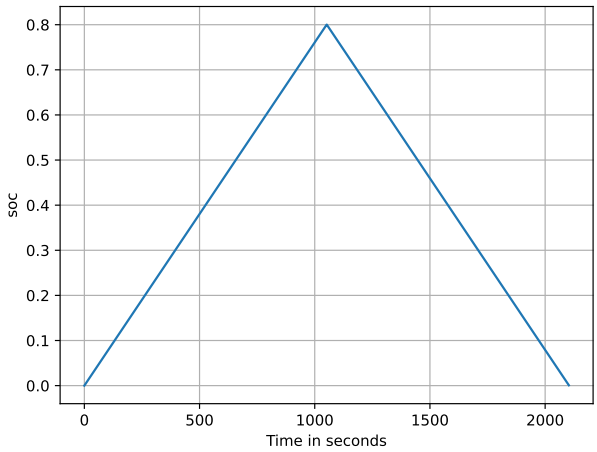


**Figure 3.5** – SOC sample from the PC schedule at 15°C, with I=1 A, mSOC=0.4 and DOD=0.8. Starting from $mSOC - \frac{DOD}{2} = 0.4 - \frac{0.8}{2} = 0$ the ECM gets charged until it reaches $mSOC + \frac{DOD}{2} = 0.4 + \frac{0.8}{2} = 0.8$ and then discharged back to $mSOC - \frac{DOD}{2} = 0.4 - \frac{0.8}{2} = 0$

**iOCV** consisted of subroutines where the ECM was subjected to a series of current pulses followed by resting phases. The pulse phase was predefined to either 30, 60, or 120 s while the relaxation time was just set to 600 s. The ECM gets first pulse-wise fully charged, then pulse-wise fully discharged.

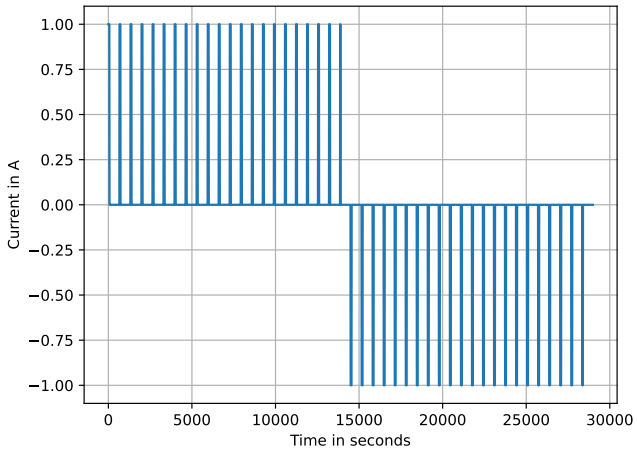


**Figure 3.6** – Current sample in the FC schedule at 15°C, with I=1 A and a pulse time pt=60 s

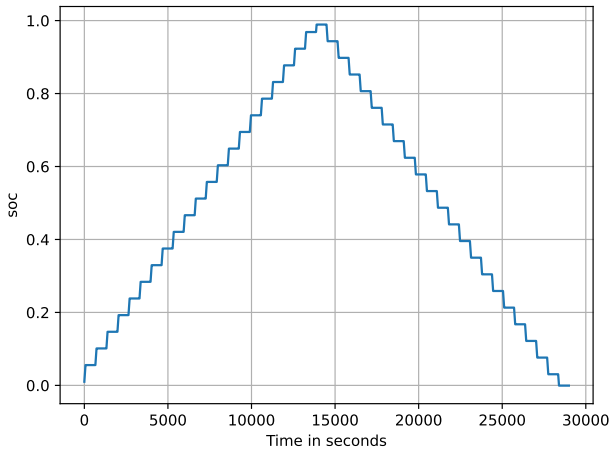


**Figure 3.7** – SOC sample in the FC schedule at 15°C, with I=1 A and a pulse time pt=60 s

All the generated data was "pickled" (stored in a binary file on the Hard drive) to train the NN models later. Stevens, Antiga, and Viehmann [4] mentioned throughout the book the importance of scaling the data passed to the NN models. For this reason, the temperature and the terminal voltage were scaled to values ranging between 0 and 1. Since the SOC data were already in that range, no scaling was applied to them. However, the current data was not scaled down to that range. The reason for

that was to stress the importance of the current values to the NN models. A high current value indicates a higher charge or discharge rate of the battery.

## 3.3 CNN implementation

The CNN implementation consisted of defining mainly 2 classes:

- **CNN_model** as mentioned in section 2.3 torch.nn is essential for building NN. The CNN class inherits from **torch.nn.Module** and redefines the methods:
  - **__init__():** to initialize the CNN architecture with the arguments
    - num_layers which represents the number of convolutional layers in the CNN architecture.
    - num_neurons which represents the number of neurons in the fully connected layer and the number of kernels. This choice was made arbitrarily. The kernel concept has been explained in subsection 2.2.2. The only difference is that we are not limited to only one kernel per convolutional layer.
    - sequence_length, which represents the length of the input sequences
    - input_keys which represents the names of the input keys. Per default, it is soc, current, and temperature. This means that the CNN gets 3 input features with a sequence length of sequence_length size each.
    - output_keys which represents the names of the output keys. Per default, it is the terminal voltage

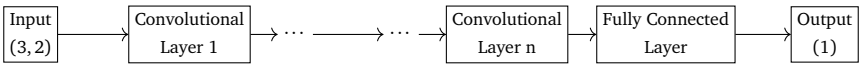


**Figure 3.8** – Chosen CNN architecture with 3 input features, each with a sequence length of 2 and 1 output feature

The final architecture of the CNN code is defined in figure 3.8. It has one input layer, num_layer convolutional layers, where the output of the final convolutional layer gets flattened and forwarded to the fully connected layer, and finally one output layer. The purpose of the CNN model is to be implemented in a BMS. For this reason, the model's size needs to be kept within reason. In this work, I prioritized the use of compact-sized models.

  - **forward()** defines how the information gets forwarded in the model. **self.convs** is an attribute containing the convolutional layers used in the architecture: the convolutional layers are created in a loop and stored

in a list called "layers". The list gets unpacked with **the operator** * and passed to nn.Sequential, which allows treating the whole container as a single module. Performing a transformation on a "Sequential" applies it to each of the modules it stores.

- **CNN_1D_dataset**: represents an iterable custom dataset. To implement it, one has to redefine the following methods:
  - **__init__** to define and initialize the attributes of the class
  - **__len__** to return the length of the entire dataset
  - **__getitem__** to return an item from the dataset given an index

  The "data" class attribute requires an input of shape (number of input features, sequence length of each feature), where all input features have the same sequence length. Additionally, a label or target is assigned to each corresponding input. The use of the dataset class is explained in subsection 3.5. To simplify this, and in the case of the generated ECM data, the CNN dataset stores all input-output pairs in a container called "data", regardless of their respective schedule.

## 3.4 LSTM implementation

Similar to the CNN implementation in 3.3, the LSTM implementation consisted of writing two classes: the LSTM class and its dataset.

- **LSTM_model**: analogous to the CNN class implementation, the LSTM class inherits from **torch.nn.Module** and redefines the methods:
  - **__init__()** to initialize the class attributes with the arguments:
    - input_size: represents the size of the input
    - hidden_size: represents number of neurons in the fully connected layer and the number of hidden units (or memory cells) of the LSTM layers. This choice was made arbitrarily.
    - num_layers: represents the number of recurrent layers in the LSTM model
    - output_size: represent the size of the output of the LSTM model
  - **forward()** to define how the information is forwarded through the LSTM model. This method defers from the one in the CNN implementation in section 3.3.

  - The input x is forwarded to the LSTM layers. These layers process the input x and the last layer returns two values: the output stored in the variable "out" and the hidden states stored in the variable "_" as it is not of interest for the next steps.
  - The "out" tensor has a shape of **(batch_size, sequence_length, hidden_size)**. We are only interested in the last time step of each sequence in the batch. For this reason, **the indexing [:,-1,:]** was used. After selection, the information is then forwarded to the fully connected layer, where it gets processed. Its output gets stored again in the same tensor "out" which gets returned at the end of the function.

- **LSTM_dataset**: there is a slight difference to how the LSTM dataset stores the data in comparison to the CNN's dataset. On top of the sequences and input-output pairs used in the CNN dataset, the LSTM dataset needs to take into consideration the order of the data. This is the reason why, in section 3.5, each generated training data was loaded separately into the LSTM dataset.

## 3.5 Training and hyperparametertuning

Training both NN models required an abstract code reusable for both models. For this, training and validation functions as well as a training loop, have been written accordingly.

The main differences between **train_model()** and **val_model()** are the optimizer and the command **model.train()** as well as **model.eval()**. The training function consists of:

- 1st load the training set which is of type CNN_dataset in the "DataLoader" mentioned previously.
- 2nd set the model in training mode with **model.train()**.
- 3rd set the training loss variable to 0 and iterate over the DataLoader.
- 4th load the input and output pairs from the DataLoader object and move them to the "device" where the computations should be happening.
- 5th set the gradient to zero. Otherwise, they accumulate after each batch, leading to incorrect parameter updates.
- 6th extract the model's output and pass both the generated output and the desired one to the loss function.
- 7th backpropagate the loss and call the optimizer

- $8^{th}$ update the training loss
- $9^{th}$ After finishing the loop, divide the training loss by the size of the used DataLoader to get the average loss per batch.

The validation function consists of precisely the same steps but with the following differences:

- **set the model to evaluation mode.** Without going into detail, the model behaves differently in the training mode than in the evaluation mode. For further explanations, please refer to the PyTorch online documentation. As a rule of thumb, one should set the model to training mode while training and to evaluation mode while validating the model or testing it.
- wrap the validation or testing loop within **torch.no_grad()** to prevent unnecessary memory to be allocated.

The training loop consists of calling the function **val_model()** and **train_model()**. But most importantly, it allows the user the choose the dataset, the optimizer, the device, the number of epochs, the batch size, the training ratio, the loss function, and if the performance should be printed or not. Those parameters are being passed to the functions **val_model()** and **train_model()** inside the loop.

Similarly to the validation function, the testing function **test_model** evaluates the model on a testing set. One could argue about using the validation function instead. However, in this case, the outputs (generated and targeted) need to be returned, as well as the loss.

Unfortunately, there is no standardized procedure for choosing the optimal parameters of the NNs. It is either with random guesses as used in section 4.3 or by using a hyperparameter optimization framework. The former is used to retrain a NN model explained in 4. The latter is used at the beginning to train the NN models. The hyperparameter optimization framework used in this thesis is "Optuna". It helps in finding good hyperparameters by intelligently searching the hyperparameter space, trying different combinations, and finally choosing the model's hyperparameters that lead to the smallest loss (explained in subsection 4.2.1).

# Chapter 4

# Evaluation

This chapter discusses the methodology on how the developped architectures were evaluated and implemented. The evaluation discussed below takes into consideration the:

- Runtime of the model (average runtime after 5 executions)
- Performance of the model using RMSE (the NN outputs were obviously unscaled before calculating the RMSE)
- Time required for preparation and implementation

After each evaluation, the best model is chosen for the subsequent comparison. Section 4.1 compares two ECM models. Section 4.2 focuses on hyperparameter tuning of the CNN and LSTM models and presents a comparison with the ECM model chosen in section 4.1. Section 4.3 presents some techniques to help increase the performance of the chosen NN in section 4.2 along with an assessment of whether the NN model in question surpassed the ECM's performances or not.

## 4.1 $1^{st}$ order ECM or $2^{nd}$ order ECM?

For this comparison, the class "Nth_Order_ECM" was used to instantiate a $1^{st}$ and $2^{nd}$ order ECMs, using the given parameters discussed in section 3.1. The $1^{st}$ order ECM didn't use the $R_2$ and $C_2$ interpolation functions in this case. Both ECMs were tested first on the charge and discharge data and finally on the FUDS dataset using the mean parameter interpolations.

Using the charge-discharge data didn't lead to a concrete answer on which ECM to use. Both were performing equally good, with a slight advantage of the $2^{nd}$ order ECM over the $1^{st}$ order ECM.

Another test had to be conducted to chose an ECM model for the next steps of this work.

Both ECM's RMSE values were estimated to 0.040763 for the 1st order one and 0.039259 for the 2nd order one. The 2nd order ECM performs a little better (by approximately 3.69%) than the 1st order ECM. The RC-circuits increase did help in the overall performance of the model in representing the non-linearities of the battery behavior. Unfortunately, this comes with a drawback in regard to increased execution time. The total time needed to perform the simulation of the FUDS dataset is shown in figure 4.1.

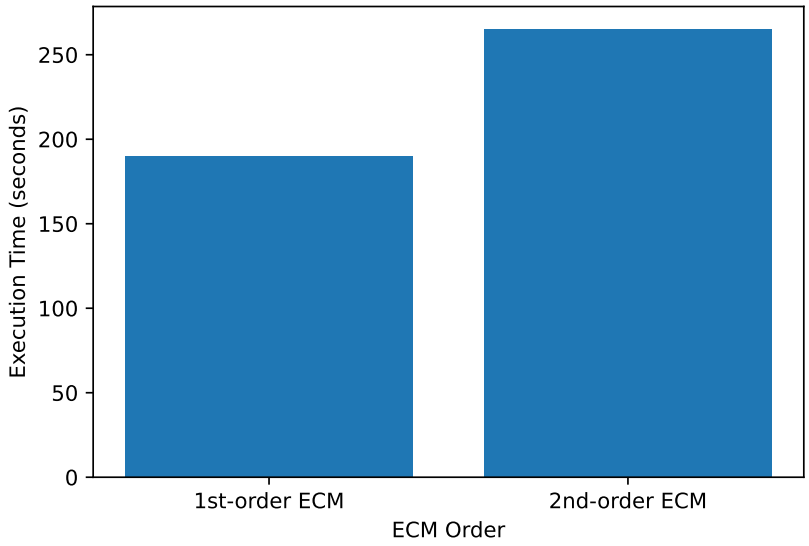


**Figure 4.1** – Average execution time comparison of a 1st and 2nd order ECM on FUDS dataset mentioned in section 3.1.

The execution time of the 2nd order ECM is roughly 38.31% higher than the execution time of the 1st ECM. This is explained through the extra matrix calculations done in the 2nd order ECM. The implementation time of both ECM models is almost the same. One problem encountered during the simulation of both ECMs was mentioned in section 3.2. Again, be careful about the choice of battery capacity and time units. The final criterium in choosing the best model would depend on the task at hand. If the purpose is to prioritize the prediction improvement, then a 2nd order ECM would be the better choice. However, if the goal is to prioritize the execution time, then a 1st order ECM would make more sense. In this thesis, a 2nd order ECM has been chosen for the next steps.

## 4.2 Comparison of ECM and NN models trained on ECM generated data

This section is divided into two subsections. In the first subsection 4.2.1, both NN models were trained on the generated ECM data as described in section 3.5 and section 3.2. The models were then compared to the chosen ECM model from section 4.1 on the charge-discharge data.

### 4.2.1 CNN and LSTM training

The optimal NN hyperparameters were found with the framework "Optuna" by giving suggested values for each of the following parameter as shown in the codes 1 and 2:

- number of layer with num_layers
- number of neurons with num_neurons
- input sequence lengths with sequence_length
- the batch size with batch_size
- the number of training epochs with epochs
- the learning rate with lr

The **suggest_categoritcal** command lets Optuna choose a value from the suggested ones. The **suggest_int** and **suggest_float**, however, let Optuna choose a value (respectively int or float) from the suggested range. The total number of trials (not shown in the Listings 2 and 1) was set to 100. Nevertheless, training NN models requires a lot of computational power and time; the search was stopped after ten trials in a row without a model improvement on the testing set. The training process can be summarized into the following steps:

- Choose hyperparameters for the NN model. This is done automatically through Optuna.
- Run the training loop mentioned in section 3.5 with the ECM generated data.
- Test the trained NN model on the charge-discharge dataset.
- Repeat the above steps until the number of trials reaches the predefined value of 100 or when the model's performance did not increase in the last ten trials in a row.

The best model is finally saved by using the command "torch.save" in combination with the state dictionary of the model.

```
def objective(trial, training_data, testing_data):
    num_layers = trial.suggest_int('num_layers', 1, 5)
    num_neurons = trial.suggest_int('num_neurons', 32, 256)
    sequence_length =
     ↪  trial.suggest_int("sequence_length",1,10)
    batch_size =
     ↪  trial.suggest_categorical("batch_size",[32,64,128,256,512,1024])
    epochs = trial.suggest_int("epochs",10,50)
    lr = trial.suggest_float("lr",1e-4,1e-2,log=True)
```

**Listing 1** – CNN: Optuna hyperparameter suggestions

```
def objective(trial, datasets, data):
    sequence_length = trial.suggest_int("sequence_length", 1,
     ↪  5)
    batch_size = trial.suggest_categorical("batch_size",
     ↪  [8,16,32,64,128])
    output_keys = ["V"]
    input_keys = ['soc','I','T']
    input_size = len(input_keys)
    hidden_size = trial.suggest_int("hidden_size", 32, 256)
    num_layers = trial.suggest_int("num_layers", 1, 5)
    epochs = trial.suggest_int("epochs", 10, 50)
    lr = trial.suggest_float("lr", 1e-4, 1e-1, log=True)
```

**Listing 2** – LSTM: Optuna hyperparameter suggestions

The optimal LSTM and CNN hyperparameters are shown respectively in tables 4.1 and 4.2.

After approximately 800 minutes of training and 13 trials, among them 10 trials in a row without improvement, the LSTM's loss was estimated to be approximately 0.000991.

| Hyperparameter | Value |
|---|---|
| Sequence Length | 4 |
| Batch Size | 32 |
| Hidden Size | 207 |
| Number of Layers | 2 |
| Epochs | 32 |
| Learning Rate | 0.000282 |

**Table 4.1** – Hyperparameters for LSTM Model

On the other hand, the CNN hyperparameters were estimated after only around 180 minutes. For this, a total of 12 trials were needed. Among them 10 without contributing to an increase in the CNN's performance which was evaluated to be around 0.000764.

| Hyperparameter | Value |
|---|---|
| Number of Layers | 4 |
| Number of Neurons | 164 |
| Sequence Length | 2 |
| Batch Size | 128 |
| Epochs | 30 |
| Learning Rate | 0.000264 |

**Table 4.2** – Hyperparameters for CNN Model

### 4.2.2 LSTM or CNN

Per analogy to section 4.1, the implemented NN models were first tested on the charge-discharge data and then on the FUDS dataset. The time performance, however, is going to be evaluated on both GPU and CPU.
Figure 4.2 displays the ECM, CNN, and LSTM RMSE performance on the charge-discharge data. The NN models were trained on the ECM generated data, aiming to approximate the behavior of the ECM model. The CNN's performance was in most cases around that of the ECM. A small difference was observed on the discharge data at 15°C.
Interestingly is, however, the performance of the LSTM model on the discharge data. It was indeed better than that of the ECM, even if it was trained on data generated by the same ECM. But it performed worse on the charge data.
Evaluating the models based on the RMSE with the charge-discharge data didn't lead to a clear answer regarding the optimal NN model choice. Consequently, the FUDS dataset was used to get a concrete answer.

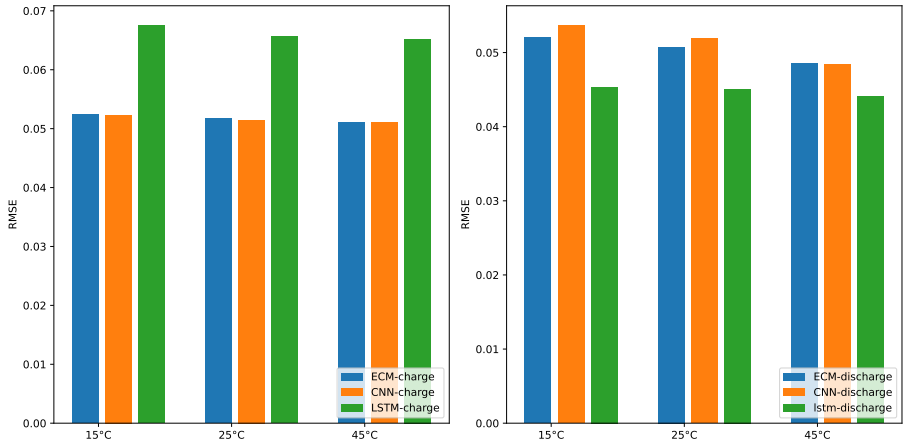


**Figure 4.2** – RMSE comparison between ECM, LSTM, and CNN on the charge-discharge data using GPU and CPU. The right plot represents the RMSE on the charge data and the left on the discharge data.

Figure 4.3 shows clearly the performance of the ECM, CNN, and LSTM models on the FUDS dataset. Because of a high RMSE value, the LSTM was disqualified from being an adequate model for the terminal voltage estimation. Subsequently, leading to comparing the CNN and ECM models only in figure 4.5.
Another reason for choosing the CNN model over the LSTM would be the required execution time, displayed in figure 4.4. The required time for estimating the terminal voltage using the CNN model is approximately three times less than that of the LSTM model. Moreover, the CNN performance on the execution time is about 12 times better than that of the chosen ECM computed on the same device (CPU).
Figure 4.5 zooms in on the RMSE performance shown in figure 4.3 by plotting the performances of the CNN and ECM models on the FUDS dataset. It is clear that the ECM still has the edge over the CNN model concerning terminal voltage prediction. In the upcoming section 4.3, we will aboard the question of improving the CNN model to make a more viable option for terminal voltage prediction.

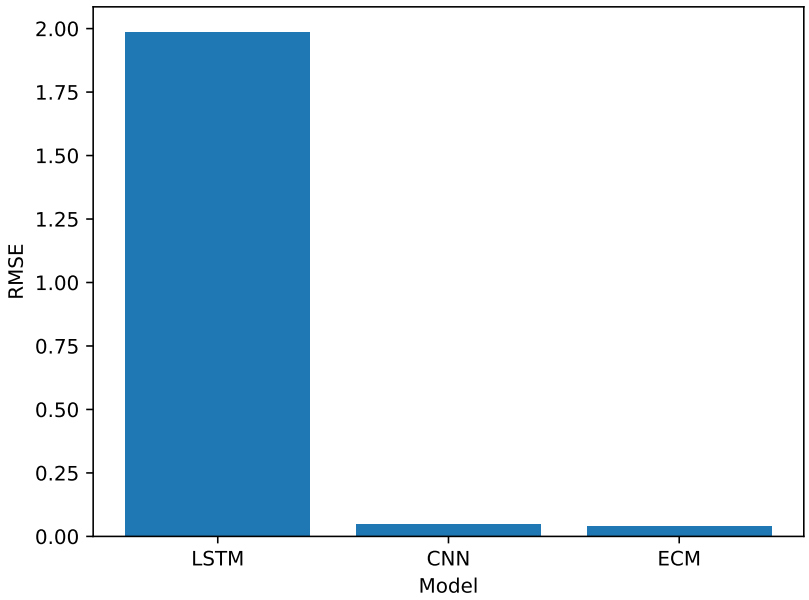


**Figure 4.3** – RMSE comparison between ECM, LSTM, and CNN on the FUDS dataset

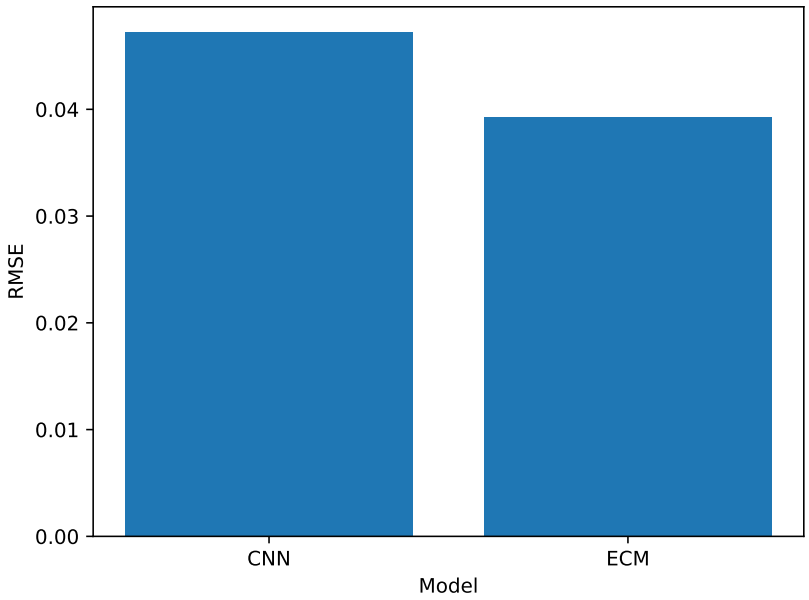


**Figure 4.5** – RMSE comparison between ECM and CNN on the FUDS dataset

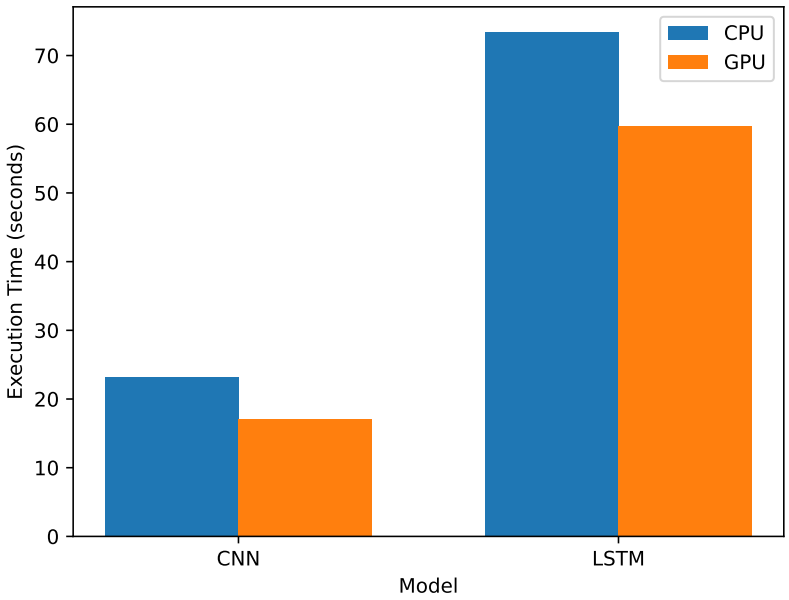


**Figure 4.4** – Execution time comparison between LSTM and CNN on the FUDS dataset

## 4.3 Improving NN model

One way to improve the CNN's performance was through fine-tuning. The training set used this time was the BMW i3 dataset described in section 3.1. Choosing the right hyperparameters this time was done through Optuna (like in section 3.5) but through trial and error (as mentioned in the same section). The CNN has been extended in the following way: The last layer of the CNN (the output layer) was replaced with a fully connected layer with 256 neurons, and after the second fully connected layer, an output layer with only one neuron for, in our case, the terminal voltage estimation. The old layers of CNN were frozen with the command **requires_grad = False** and the new layer of the CNN was trained on the BMW i3 dataset. Using all the available datasets led to overtraining: the model could no longer generalize on the new unseen data. To resolve this issue, I had to resort to choosing only one dataset for the training: the **TripB01.csv**.
The final performance of the fine-tuned CNN is shown in figures 4.6, 4.7 and 4.8. The CNN's decreased a little compared to its performance before fine-tuning. Moreover, adding a new fully connected layer increased the average runtime using the CPU by 5 s. While its runtime on GPU stayed unchanged.

The estimated time for finding the suitable dataset that enhanced the prediction performance of the CNN took me less than an hour. This means using a hyperparameter optimizer like Optuna is not always the best solution.
We can conclude that NN models trained on ECM-generated data can't surpass the ECM in question.

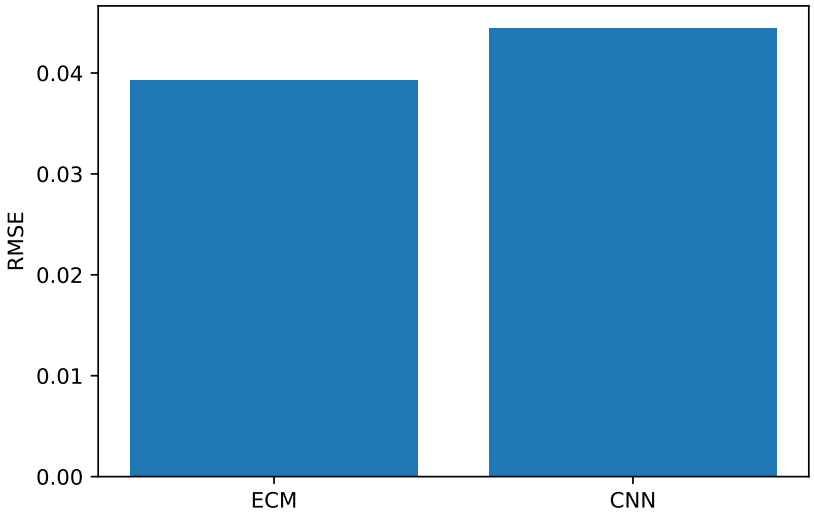


**Figure 4.6** – RMSE comparison between ECM and fine-tuned CNN on the FUDS dataset

## 4.4 Implementation time complexity

The most time-consuming part of this thesis was the implementation of the ECM based on the library **PyBaMM**. It took three weeks until I realized that the ECM I wanted to implement was impossible with that library, thus leading me to implement my own ECM class discussed in section 3.2. This took me three days in total. Writing the NN classes discussed in sections 3.3 and 4.2.2 took me a few weeks. The reason for this was the fact that I wanted to test my hardware capabilities concerning implementing NN models. However, I got conscious of the purpose of the NN models in this work. There was no reason to implement a CNN with millions of neurons per layer (in the fully connected layers) with multiple fully connected layers. Instead, I changed my focus to implementing optimal and small NN models for the task at hand. If the NN models still couldn't surpass the ECM performances even after training on the BMW i3 data, they could at least surpass it in terms of execution time. This took me a total of one and a half months. Running some tests on the trained NN models

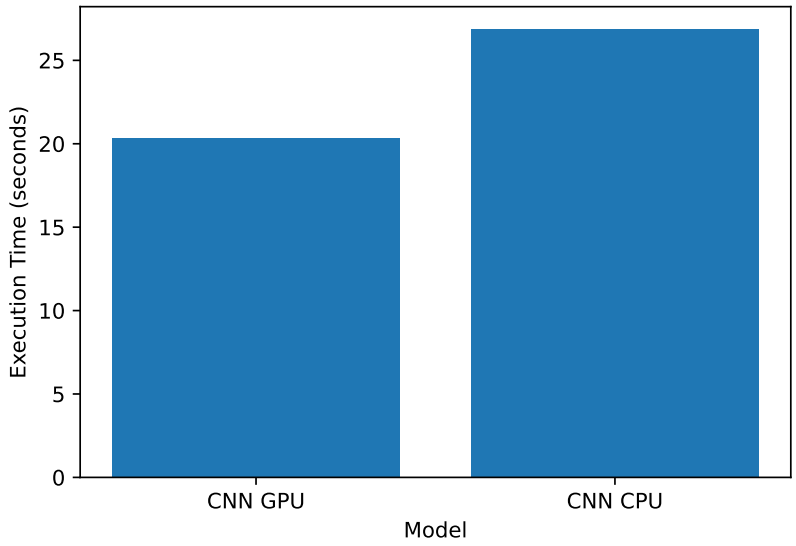


**Figure 4.7** – Fine-tuned CNN average execution time comparison on FUDS dataset using CPU and GPU

with new other datasets is straightforward: One has to first keep in mind that the input features must be in the following order: SOC, Current, and Temperature. The SOC values should range between 0 and 1 (not in percentage form). The current doesn't require scaling. The temperature, however, does. These features must be stored in a custom dataset: either LSTM_dataset or CNN_1D_dataset. Finally, the dataset must be passed to the test_model function, along with the model to be tested.

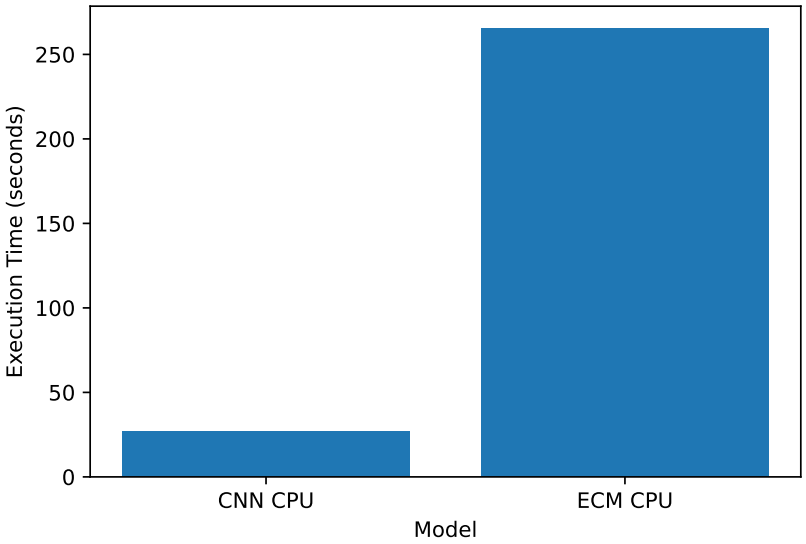


**Figure 4.8** – Average execution time comparison between ECM and fine-tuned CNN on the FUDS dataset using CPU only

# Chapter 5

# Conclusion

The work done in this thesis involved selecting an ECM, generating data, and evaluating the performance of NN models for battery monitoring tasks, specifically for terminal voltage estimation. The necessary software for the simulation has been developed using Python programming language with the framework PyTorch. The resulting software allows the user to comprehend the strengths and limitations of the models for the above-mentioned task. The ECM class was implemented, and two ECM models were instantiated of respectively $1^{st}$ and $2^{nd}$ order, and compared in terms of prediction and runtime performance. The $1^{st}$ order ECM offered better runtime performance in comparison to the $2^{nd}$ order ECM. However, the latter was found to offer a slightly better prediction performance than that of the former. Subsequently, training data was generated using the $2^{nd}$ order ECM for the NN models.

It was identified that the CNN and the LSTM were adequate for the task at hand. After training, the performance of the CNN was found to be superior to the LSTM in terms of prediction accuracy, with similar performance compared to the ECM model. Additionally, the CNN's runtime exceeded that of the LSTM and, by far, that of the ECM. The problem of the CNN's similar prediction performance in comparison to that of the ECM couldn't be tackled by fine-tuning the already trained CNN. The CNN's last layer was changed with a fully connected layer, followed by a new output layer. The already-trained layers were frozen, and the new layers were trained on non-ECM-generated data: the BMW i3 driving dataset. This dataset was scaled down to the battery specification of this thesis and used for the training of the newly implemented CNN layers. These enhancements resulted in a small improvement in prediction performance without surpassing that of the ECM model.

The results of this work demonstrate that NN models, in general, and CNN, in particular, are viable alternatives to ECM for battery modeling. The drawback of using NN is that they need a lot of data for training. This would take quite some time

if one were to run the same scheduled tests directly on the battery. This is where the ECM comes in handy; instead of conducting experiments directly on the battery, one could use the ECM to generate those data very fast and finally feed them to the NN models

Various aspects of this work could be improved in the future. The developed code should implement more error messages in case the user's input differs from what the code is intended to do. The LSTM should be investigated further. Besides, the developed NN models suffer from the problem of being sensitive to the input feature's order. It would be better if the input features could somehow be reorganized or at least checked in the case that their order differs from that of the initial training. Finally, the battery state estimation should not limit itself just to terminal voltage estimation but also to other states like temperature, aging, etc.

# List of Figures

# List of Tables

# Bibliography


[1] G. L. Plett, Battery management systems: battery modeling. Volume 1, en. Boston : London: Artech House, 2015, OCLC: ocn909081842.

[2] M. A. Samieian, A. Hales, and Y. Patel, "A Novel Experimental Technique for Use in Fast Parameterisation of Equivalent Circuit Models for Lithium-Ion Batteries," en, *Batteries*, vol. 8, no. 9, p. 125, Sep. 2022. DOI: 10.3390/batteries8090125.

[3] M. Ruba, R. Nemeș, S. Ciornei, and C. Marțiș, "Parameter Identification, Modeling and Testing of Li-Ion Batteries Used in Electric Vehicles," en, in *Applied Electromechanical Devices and Machines for Electric Mobility Solutions*, A. El-Shahat and M. Ruba, Eds., IntechOpen, Mar. 2020. DOI: 10.5772/intechopen.89256.

[4] E. Stevens, L. Antiga, and T. Viehmann, Deep learning with PyTorch, en. Shelter Island, NY: Manning Publications Co, 2020.

[5] J. Patterson and A. Gibson, Deep learning: A practitioners approach. OReilly, 2017.

[6] R. S. Ransing, M. R. Ransing, and R. W. Lewis, "On the limitations of neural network techniques for analysing cause and effect relationships in manufacturing processes - a case study," en, vol. 29, 2003.

[7] D. Pedamonti, Comparison of non-linear activation functions for deep neural networks on MNIST classification task, en, arXiv:1804.02763 [cs, stat], Apr. 2018.

[8] F. Chollet, Deep learning with Python, en. Shelter Island, New York: Manning Publications Co, 2018, OCLC: ocn982650571.

[9] X. Hu, S. Li, and H. Peng, "A comparative study of equivalent circuit models for Li-ion batteries," en, *Journal of Power Sources*, vol. 198, pp. 359–367, Jan. 2012. DOI: 10.1016/j.jpowsour.2011.10.013.

[10] X. Zhang, W. Zhang, and G. Lei, "A Review of Li-ion Battery Equivalent Circuit Models," en, *Transactions on Electrical and Electronic Materials*, vol. 17, no. 6, pp. 311–316, Dec. 2016. DOI: 10.4313/TEEM.2016.17.6.311.

[11] S. Jiang, "A Parameter Identification Method for a Battery Equivalent Circuit Model," en, Apr. 2011, pp. 2011–01–1367. DOI: 10.4271/2011-01-1367.