CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Reinforcement Learning-Based Cell Balancing for Electric Vehicles

Master's thesis in Computer science and engineering

GIOVANNI MAZZOLO

MATEI SCHIOPU

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Master's thesis 2024

Reinforcement Learning-Based Cell Balancing for Electric Vehicles

GIOVANNI MAZZOLO

MATEI SCHIOPU

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2024

Reinforcement Learning-Based Cell Balancing for Electric Vehicles

GIOVANNI MAZZOLO
MATEI SCHIOPU

Supervisor: Dr. Yang Xu, Volvo Group
Advisor and Examiner: Pedro Petersen Moura Trancoso, Department of Computer Science and Engineering

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2024

Reinforcement Learning-Based Cell Balancing for Electric Vehicles

GIOVANNI MAZZOLO
MATEI SCHIOPU

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Lithium-ion battery packs are comprised of hundreds to thousands of individual cells which, even though manufactured uniformly, exhibit small variations in their characteristics that impact their behavior during operation. These differences cause cells' State of Charge (SOC) to become unbalanced, which can, in turn, reduce the capacity utilization efficiency of the pack [1]. Additionally, battery cells age differently over time, and fast-aged cells can cause packs with healthy cells to be retired early, without fully taking advantage of each cell. When a battery has deteriorated to around 80% of its total capacity, it is retired from electric vehicle usage [2].

To maintain batteries functioning correctly, cell SOC balancing must be done on battery packs. However, balancing the SOC of cells provides a window of opportunity to also include cells' health into the balancing equation, aiming for the homogenization of cell aging, allowing to thoroughly utilize a battery's resources. In this way, it is possible to both keep batteries in operating condition and potentially increase their lifespan.

In this work, we develop and research a multi-cell simulation framework and Reinforcement Learning (RL) methodologies to explore the potential of cell SOC and health balancing. We propose an active balancing strategy for re-configurable cell topology with RL, in which instead of transferring energy between high SOC cells to low SOC cells, cell utilization is modulated so that the power consumption is optimally distributed based on each cell's SOC. This strategy is applied to SOC balancing, as well as SOC and State of Health (SOH) balancing simultaneously, to potentially allow for an exhaustive utilization of the battery's potential.

Keywords: Battery, cell balancing, reinforcement learning, lithium-ion batteries, automotive, computer science, engineering, deep learning.

Acknowledgements

We would like to express our deepest gratitude to our supervisor Dr. Yang Xu during our thesis project at Volvo Group, for always showing kindness, patience, willingness to help, and prioritizing assisting us on the project even during busy days. He helped us broaden our views, offered advice, and motivated us to do our best and to continue learning. Without his great experience, knowledge, and commitment to help, this work would not have been possible nor have been as enjoyable as it was. Even when at times we made mistakes or when we were not at our best, he made sure to always stay positive and supported us to keep moving forward. We sincerely thank you.

Many thanks to the Volvo BMS team for the feedback, and guidance and for welcoming us to the team during our thesis work. Thank you for the amazing insights into the world of lithium batteries.

We would also like to thank our Chalmers supervisor, Pedro Petersen Moura Trancoso for supporting us during the thesis work.

This Master's thesis was conducted at Volvo GTT in conjunction with Chalmers University of Technology, Department of Computer Science and Engineering.

Giovanni Mazzolo and Matei Schiopu, Gothenburg, 2024-06-26

Contents

Acronyms

Below is the list of acronyms that have been used throughout this thesis:

BMS	Battery management system
RL	Reinforcement learning
ML	Machine learning
AI	Artificial intelligence
EV(s)	Electric vehicle(s)
SOC	State of charge
SOH	State of health
SOQ	State of capacity
SOR	State of resistance
SOX	State of charge, health, capacity, resistance
DOD	Depth of discharge
ET	Energy throughput
PPO	Proximal policy optimization
SAC	Soft Actor-Critic
DPG	Deep Policy Gradient
DDPG	Deep Deterministic Policy Gradient
TD3	Twin Delayed DDPG
OCV	Open circuit voltage
DQN	Deep Q-Network
RC	Resistor–capacitor
PID	Proportional–integral–derivative
MPC	Model Predictive Control
ADAM	Adaptive Moment Estimation

gSDE	Generalized State-Dependent Exploration
EM	Electro-chemical Models
ECM	Equivalent-circuit Models
DDM	Data-driven Models
UCB	Upper Confidence Bound
RUL	Remaining-useful-life
BPNN	Back Propagation Neural Network
RBNN	Radial Basis Neural Network
LSTM	Long Short Term Memory
ANN	Artificial Neural Network

Nomenclature

Reinforcement learning variables

γ	Discount factor
π	Policy
σ	Logical sigmoid function
$ET(s, i)$	Energy throughput of a cell in a certain state
$R(t)$	Return. Total sum of rewards starting from timestep t onwards, modified by discount factor
r_t	Reward assigned at timestep t

Other symbols

C_j	Balancing feedback for power modulation

Physics notations

η	Coulombic Efficiency
Ah	Ampere hour (Amp $\times$ hour)
$C - rate$	Charge rate
$i(t)$	Current
$i_{app}(t)$	Discharge or charging current
$i_{balance}(t)$	Balancing current
$i_{leakage}(t)$	Leakage current
$i_{net}(t)$	Total current
$i_{self-discharge}(t)$	Self discharge current
P	Power
Q	Capacity
q_i	Cell capacity
R	Resistance

R_0	Cell R_0 resistance
$v(t)$	Voltage
$v_{OC}(t)$	Open Circuit Voltage
$v_{RC}(t)$	RC circuit Voltage
$v_t(t)$	Terminal Voltage
$z(t)$	State of charge
$i_{RC}(t)$	RC circuit current

List of Figures

List of Tables

1

Introduction

Electric vehicles (EVs) have seen a massive surge in popularity over the past years, with reports from the International Energy Agency showing over 26 million EVs on the road in 2022, with more on the rise [3], giving way to a sustainable transportation method for people and industries. The increase in demand for electric vehicles goes hand in hand with battery demand, which comes with its own unique set of challenges. The battery pack comprises the most expensive component in EVs, with prices of $\$200/\text{kWh}$ [4]. Once the capacity of a battery degrades to a certain level, around 80%, the pack is deemed expired and must be replaced. The preservation of battery lifetime is thus integral and highly desirable in the field. Maximizing the amount of charge within a battery pack is yet another important research subject within battery technologies, to grant vehicles as much range as possible.

Cell balancing is the process of bringing all cells in a pack to similar charge levels. Both battery lifetime and performance depend on the balancing mechanism of the battery. A lack of balancing will quickly render a battery unusable [5]. The widely-used industry balancing applications are using relatively simple topologies using control algorithms that are not battery health-aware and cannot effectively make use of all of the sensor data at their disposal. Traditional control mechanisms that efficiently follow multiple balancing objectives are difficult to develop, as they rely on precise modeling of battery pack relationships [6] [7]. Due to the non-linear and estimative nature of battery dynamics, when multiple objectives are pursued, the complexity of developing such control systems grows exponentially.

Traditional balancing topologies rely on passive balancing, which wastes the charge from the cells, or active balancing, which attempts to transfer energy between cells until they are balanced. Both of these balancing methods happen during resting periods when the battery is not in use [8].

Within this thesis, we aim to introduce a cell balancing method that functions during vehicle operation by modulating cell utilization via reconfigurable cells according to the charge and capabilities of each cell. We develop a reinforcement learning (RL) balancing controller for this topology, training it to handle the complexity of such a topology, and attempt to balance the State of Charge (SOC) in a health-aware manner, preserving the State of Health (SOH).

1.1 Research objectives

We propose a new reconfigurable cell balancing topology, which aims to balance cells during operation. Our proposed topology utilizes a reconfigurable battery solution alongside power electronics that allows balancing to take place during cell operation, introducing the possibility of balancing outside rest phases - which is the norm for current battery management systems. Such a topology requires a complex controller that can interface with it and follow the balancing objective.

RL agents can infer the required control actions without the requirement of relationship modeling used in methods such as MPC (Model predictive control) [7], while also being able to provide strong predictive behavior. This topology makes use of a high-dimensional continuous action space and observation space, in a highly non-linear environment, a challenging RL task, and we aim to provide an RL balancing controller that can effectively train to handle such an environment, as well as attempt to follow multiple training objectives.

The aim of this thesis is to introduce a reinforcement learning approach to cell balancing. The intended outcome is to create a cell balancing controller built from a reinforcement learning agent, as well as provide the simulation environment for the training and testing of the agent. Designing a balancing controller that can take many factors into account using traditional control theory methods, for a non-linear environment, gets exponentially harder with each objective as there are difficulties in modeling these non-linear relationships between each new factor. We aim to train an RL controller agent to learn these relationships and make use of them in battery balancing, and simultaneously pursue multiple objectives such as SOC balancing in a health-aware manner.

1.2 Related work

With the rise of electromobility, research in the area of battery systems has expanded rapidly. The first component of battery management systems (BMS) to be interfaced with AI/ML technology has been parameter and state estimation and prediction. Several papers discuss the applications of supervised and unsupervised learning in order to predict the values of SOC, SOH, capacity, battery aging, and other functioning parameters. There have been extensive studies targeting the estimation of SOC, SOH, and Remaining-useful-life (RUL) for battery systems using machine learning. However, the cell balancing aspect of BMS remains an open question in the field of machine learning, specifically when it comes to exploring the capabilities of reinforcement learning for battery controllers.

Harwardt et al. [9] consider a reinforcement learning method that targets passive and active cell balancing using a cell simulation that contains a dynamic thermal model. However, this method is limited to balancing one cell per timestep and making powerful assumptions, such as constant voltage.

Y. Yang et al. [10] explore a fast-charge RL battery control method using DQN for minimizing charging time in a balancing-aware manner. By using the generalization of

a neural network, the RL agent handles balancing control during charging.

Duraisamy et al. [11] propose cell balancing methods that utilize back propagation neural network (BPNN), radial basis neural network (RBNN), and Long Short-Term Memory (LSTM) models to select an optimal resistor for passive battery balancing. Each model's parameters are based on SOC, temperature rise, balancing time, and C-rate and they are compared and evaluated on a 3-cell and 3-resistor simulation environment. The study treats the case of the switched shunt resistor passive balancing topology and proposes a set of additional resistors that the model can swap between at will, comparing it to the limited capabilities of a weaker topology.

Z. Xia et al. [12] utilize an artificial neural network (ANN) to estimate the battery capacity, which then feeds into a controller to manage the balancing of the battery cells. Both of the presented studies rely on machine learning techniques only for battery measurements, estimating their states.

B. Jiang et al. [13] study a Deep Neural Network RL method of controlling a reconfigurable cell topology based around switches, using a discrete action space. This balancing method considers the only SOC of cells for the RL agent's balancing objective.

The technical basis for battery pack modeling and simulation methods used for developing the new topology is adapted from G. Plett's publications [14] [8]. These works provide exhaustive descriptions of physics-based models of lithium-ion cells and state-of-the-art applications of equivalent-circuit models used for battery management and control. The battery pack simulations of the project were adapted from these works.

The reinforcement learning algorithms studied within this thesis and used to develop the RL controller are based on the following works. T. Haarnoja, et al. [15] describe Soft Actor-Critic (SAC), an off-policy actor-critic deep RL algorithm based on the maximum entropy reinforcement learning, which is utilized in our work to train RL controllers. S. Fujimoto et al. [16] propose a novel mechanism that builds on Double Q-learning, by taking the minimum value between a pair of critics to limit overestimation, and results in the Twin Delayed Deep Deterministic policy gradient algorithm (TD3) which we experiment with in our work. T. P. Lillicrap et al. [17] define the Deep Deterministic Policy Gradient, the actor-critic, model-free algorithm based on the deterministic policy gradient that can operate over continuous action spaces, and the principles that would later serve as the basis for TD3 and SAC algorithms. J. Schulman et al. [18] propose Proximal Policy Optimization (PPO), policy gradient methods for reinforcement learning which alternate between sampling data through interaction with the environment, and optimizing an objective function using stochastic gradient ascent.

Existing studies treat cases of 3 cells [9], [11] or 4 cells [10], while others utilize machine learning only for estimation rather than direct control [12]. We seek to provide a scalable RL methodology that introduces state-of-the-art RL algorithms to the field and provides insight into RL development methods in the context of EV battery controllers. The existing methods handle balancing during resting [9], [11], or charging [10] phases. A study [13] proposes a reconfigurable topology based around switches, a discrete action space, coupled to an RL controller which only takes SOC into consideration in the reward function. We introduce a power-electronics battery balancing topology, with a

continuous action space and high-dimensional continuous observation space, which can be active during battery discharge during normal operation of a vehicle, coupled with an RL controller capable of handling this continuous action space to balance SOCs as well as maintain SOH through health-aware balancing.

1.3 Methodology

Within our work, we propose and analyze applications of reinforcement learning agents for battery pack balancing control, as a method to follow multiple objectives within balancing. This is done within a dynamic battery pack simulation based on experimental cell datasets. Within the project, we adapt the simulation formulas to our proposed topology and link it up to simulated road-usage powder demands profiles, synthesized from a drive-cycle dataset, and altered by road conditions. This simulation is then attached and synchronized with reinforcement learning algorithms within our training framework, in order to train and produce RL agents which can fulfill the role of controllers for the BMS of the EV battery in an online RL way. We undergo a thorough analysis and experimentation of reinforcement learning algorithms, training techniques, optimizations, and reward design in order to find the most suitable algorithm for training RL agents for battery pack environments and provide insight into development methods for the complex high-dimensional and non-linear learning task of health-aware battery balancing.

1.4 Organization

The thesis is formatted into several chapters. The theoretical background and foundational information are presented in the 'Background' chapter, which is split into the battery pack, balancing, and RL theory. The implementation and experiments are discussed in the 'Methodology' chapter which presents the work done on the cell simulation, RL algorithms, and the interfacing of the two. The findings are showcased in the 'Results' chapter. Discussion based on the findings and final thoughts for applications are presented in the 'Conclusion' chapter, as well as proposals for future improvements and subsequent work.

1.5 Limitations

The electrical and electronics design aspect of the topology makes use of abstractions, especially in the case of duty-cycling power electronics. In-depth design and simulation of these elements are not within the scope of the work. Training of the RL model for this project was done using cell simulations that are based on the characterization of real cells, which are modeled using a first-order RC circuit, which we consider sufficient for the needs of this project.

1.6 Ethical Considerations

All datasets used within this work for cell modeling and drive cycles are public and contain no identifiable vehicle data. Reinforcement learning models, especially Deep Learning variants, are highly complex and raise difficulties in interpreting their decisions. This lack of transparency can lead to issues for safety considerations. Any EV applications that attempt to incorporate RL/ML technologies within their workflow must undergo extremely thorough testing and verification, especially in the case of battery technologies which can be subject to leaks of hazardous material or smoke, fire, and explosion as a result of thermal runaway.

2

Background

2.1 Battery Management Systems

Battery management systems (BMS) are software applications that oversee the safe and efficient operation of battery cells and the battery packs as a whole [19].

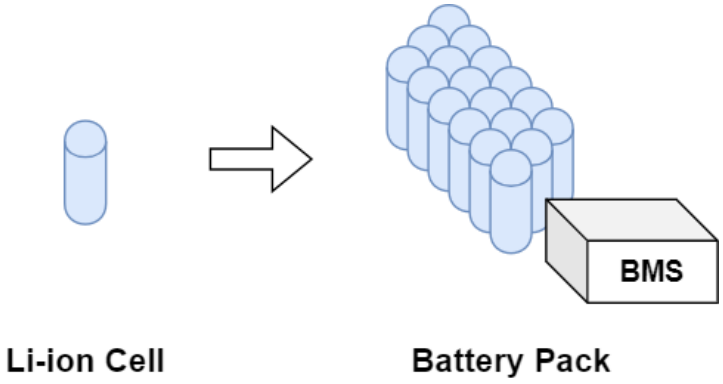


Figure 2.1: The composition of EV battery packs

A battery pack can contain hundreds of cells, and an electric vehicle (EV) can contain multiple battery packs, pictured in Figure 2.1. Traditionally, EV manufacturers use the same cell type in vehicle packs, as mismatches add unnecessary complexity to functioning and manufacturing. However, cells of the same type, from the same manufacturer and manufacturing batches are not identical in characteristics. Current manufacturing limitations and industry demand do not allow precise uniformity among the cells [14] [8].

A BMS is responsible for keeping track of data from sensors across the battery pack, such as voltages, current, and temperatures, and uses it to maintain uniform usage in a non-uniform environment. Some sensors are present at cell-level, on each cell, such as the ones for current and voltage, while some can be placed across several sections of the battery pack in the case of thermal sensors. This data is used to manage temperature, charge C-rate, discharge profile, and other functions of the pack [19].

The raw data from the sensors is then used to estimate and predict 'states' for the cells. State of Charge (SOC), State of Health (SOH) (including State of Capacity (SOQ) and State of Resistance (SOR)), and many other states are estimated by the BMS to determine the status of the battery pack. SOC, for example, can be used to directly

inform the power management unit of the remaining charge and, coupled with additional vehicle usage data, determine the remaining range of the vehicle and offer an estimation of how much longer the vehicle can run during assumed average usage [14].

2.2 State of Charge (SOC)

Arguably the most important estimation performed in the battery back is the State of Charge. It is crucial to the basic functionalities of a vehicle, however, SOC has many more important uses that are not directly visible to the end-user of an EV. As we cannot directly measure SOC, we must rely on estimation methods [14].

Rechargeable lithium-ion cells are small packets of different shapes and sizes that store electricity through chemical reactions. Cell usage is represented by charge/discharge rate, known as cycling rate (C-rate), and voltage, must be within manufacturer-defined limits [14].

State of Charge is bound to these limitations. If a cell were to over-charge beyond its factory limit, chemical reactions within the cell would lead to irreversible damage and a permanent loss of capabilities in the cell. The same goes for over-discharging. A cell is never 'fully discharged' of its electric charge, but rather partially discharged up to a lower voltage limit which it must never pass [19]. When discharging, this usage bound is called Depth of Discharge (DOD), and it measures how much of a battery's capacity has been used relative to its total capacity. DOD has a direct impact on the degradation of a cell, as higher DOD generally results in a larger volume change of active particles during cycling, increasing stress and leading to cracking and cell degradation [20].

The definition of SOC varies depending on the method used for the estimation. The most widely used methods are Open circuit voltage, Coulomb counting, Impedance Spectroscopy, Model-based or ANN-based.

In the Open Circuit Voltage (OCV) method, SOC is defined as a one-to-one relationship to OCV. The battery is disconnected from the load and left to settle its chemical reactions, and then OCV is measured. It is then compared to a SOC-OCV lookup table and the SOC is determined [21]. This method however can require long relaxation times until a lithium-ion battery is fully settled. The lookup table depends on the battery chemistry, and the relationship between SOC-OCV can diverge from the one-to-one estimation with temperature changes and aging.

In the Coulomb counting method, SOC is defined as the integration of current [21]. Given an accurate initial SOC estimation, coulomb counting keeps track of the current that flows in and out of the battery by accumulating the charge that is being transferred [22]. However, this method relies on the accuracy of the counting sensors, and inaccuracies from defects or electronics wear can accumulate into large estimation errors [21].

Electrochemical impedance spectroscopy functions by injecting small amplitude AC signals to a battery at different frequencies. Parameters measured from these signals are then used as indicators of SOC. This method is not suitable for online measurements, while the battery is being used, [21] and usage is mostly restricted to lab environments.

The model-based methods are the most suitable for online measurements and involve modeling of the electrical, chemical, or combinations of both properties of a specific battery. The most widely used algorithms for this method are variations of the Kalman filter [23].

For our project, we utilize coulomb counting to track SOCs. We aim for a simulation of a battery within a vehicle in usage, rather than within a lab environment, and thus we make use of an accurate online, real-time, measurement method.

2.3 State of Health (SOH)

The State of Health (SOH) is another critical parameter for BMSs that aims to estimate the loss of charge storage capabilities of a battery, by use and aging [24]. Fundamentally, it is the monitoring of the battery's parameters whose degradation process is, usually, slow yet influences the battery's performance.

Over time, a battery's total capacity gets reduced - this is known as capacity fade. Capacity fading occurs due to the cell's structural deterioration as well as chemical side effects over time. Furthermore, battery cell aging causes its internal resistances to increase, for similar reasons. It is essential to have an accurate measurement of the capacity and internal resistances, given that they are major contributing parameters for estimating SOC and calculating energy [14].

Similar to SOC, SOH is also a parameter obtained through estimation and several methods enable it, such as [24]: Characteristic Quantity Method, Model-based Method (Equivalent Circuit Model, Electro-chemical Model, Electro-chemical Impedance Spectroscopy), Fusion Method and Data-driven Methods (Neural-networks, Fuzzy Logic, Support Vector Machine, Parameter Identification).

An influential aspect of a battery's health is the C-rate, which defines how quickly the cell acquires or releases charge. C-rate has many implications on a cell's lifetime and capacity, the most important one being that high C-rates lead to drastic loss of capacity and loss of power for the cell [8] [20]. Essentially, batteries that are under extreme load demands or that are charged at a very high rate can exhibit faster degradation, leading to reduced lifetime.

Many elements are considered to influence the degradation of SOH in a battery during operation. The most drastic effects arise from overcharging and over-discharging, as well as operation at inadequate temperatures [8].

Within our work, we treat these effects via charge/discharge safety mechanisms and temperature abstractions within the battery simulation, and focus on the effects of cell balancing and cell usage. We use energy throughput as an indicator for preventing SOH degradation. By attempting uniform energy throughput for each cell while balancing, we aim to decrease the overall degradation of cells and limit the non-uniform degradation of cells in a pack to increase the lifetime of battery packs.

Some state-of-the-art work [12], [25] that include SOH in the balancing strategy model SOH as the capacity fade of the cells. However, capacity fading can be caused by

several factors, including internal chemical reactions and cell operation. Cell chemical deterioration is not considered for this work as it is unavailable in our simulation capacity or dataset. For this, we consider SOH as the energy throughput of the cells. Cell throughput when considered as SOH allows for the RL model to be trained on data that updates consistently and in a fast manner, in addition to energy throughput being linked to cell aging [20].

2.4 Causes of imbalance

Imbalance is introduced by any combination of factors that make the SOCs of the cells diverge from one another. A common factor is the different Coulombic efficiency in cells. Two cells may start in the same SOC state, and quickly diverge during charging due to the different efficiency η [8].

The following formula describes how the SOC z of a cell is estimated while the cell is being charged

$$z(t) = z(0) - \frac{1}{Q}\int_0^t \eta(\tau) i_{app}(\tau) d\tau \tag{2.1}$$

where Q is the capacity, η is the Coulombic efficiency and i_{app} is the charging current of the cell. Notice that η directly affects the charging current of the cell, and each cell's η parameter is unique, even a slight difference can cause a cascading unbalancing effect over time.

Imbalance can also happen due to different total current loads applied to the cells. The current that passes through cells is related to multiple different sources. Besides the main application current, the load requested from the battery pack for the operation of a vehicle, we also have self-discharge current as well as leakage current [8].

$$i_{net}(t) = i_{app}(t) + i_{self\text{-}discharge}(t) + i_{leakage}(t) \tag{2.2}$$

Self-discharge rate differs from cell to cell and refers to the internal current flow within a battery cell when it is not connected to any external circuit. Self-discharge is a natural phenomenon that occurs in all types of batteries over time due to ongoing chemical reactions within the battery, even when not in use, as well as impurities and imperfections in manufacturing which create small pathways for current to flow internally. These chemical reactions accelerate with temperature increases, as well as high SOC values within the cell over long periods [8].

Leakage current refers to the small amount of current that powers attached BMS electronic circuitry. Manufacturers of battery packs and BMS components specify the power consumption of their BMS circuitry in datasheets [8].

The effects of these additional currents are seen through every state of the battery, in charging, discharging, and resting. They are permanently active and vary with the parameters of the pack and cell itself [8].

2.5 Cell Balancing

The definitions of a balanced pack differ from application to application. A balanced pack is commonly defined as a pack where all of the SOCs of its cells are the same. However, this is a very harsh requirement that depends strongly on very accurate measurements and cell manipulation techniques which are not usually available in an EV system. The SOCs in an EV battery pack vary greatly during usage and the estimations from measurements can be inconsistent, as the cells are not measured in an isolated lab environment [21].

Thus we introduce a balance threshold rather than an exact equality relationship. In a cell pack with SOC values between 0 and 100%, we compute the difference between the smallest SOC value and other cells in the pack. We assign a threshold for the maximum SOC difference between the cells, which within experiments can take values between 1% to 3.5%. Cells within the pack with a difference smaller than the set threshold are considered balanced.

Due to the non-uniform characteristics of cells in a pack, which diverge even further through usage and aging, cells in a pack begin to have very different SOCs. As the cells are charged and discharged, these SOC variations get larger and larger to the point where, for example, a cell can be at below 50% SOC when the rest are almost at 100%. The short-term effects are a significant reduction of battery capacity and a waste of energy, reducing the range of an EV to a fraction of the intended baseline.

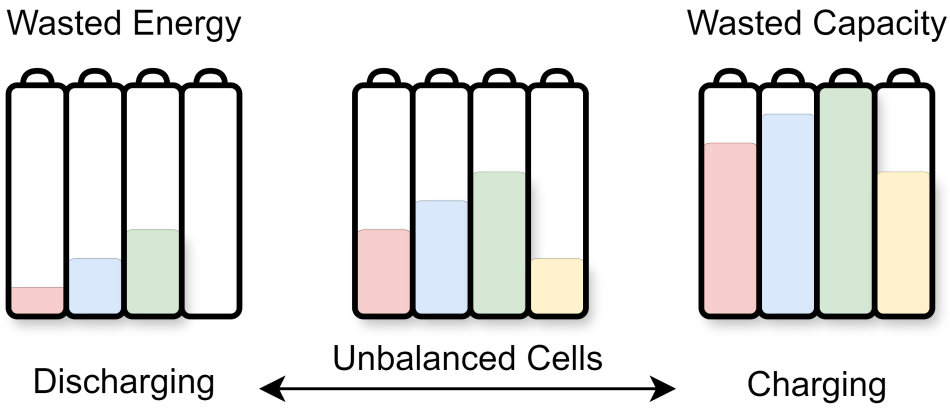


Figure 2.2: The immediate effects of cell imbalance

When such differences happen, a BMS limits the battery capacity to that of the weakest cell in the pack, pictured in Figure 2.2. Continuous usage of the vehicle with these imbalances can lead to rapid deterioration of the cells, rendering the whole pack unusable. An improper BMS implementation on an EV suffering from cell imbalance can lead to a full stop in the middle of a highway and similar safety risks [8]. These imbalances, if left untended over a longer period, lead to rapid degradation of the battery cells through accelerated aging and abuse. Erroneous charging/discharging of imbalanced batteries, usually due to measurement inaccuracies of batteries during stress, leads to overcharge

and over-discharge. This produces irreversible chemical reactions in the batteries, which reduce their capabilities and generate heat [5].

Imbalance in cells influences internal resistance. Cells with higher internal resistance tend to dissipate more heat during charging and discharging. This increased heat generation can exacerbate temperature differences within the battery pack, potentially leading to localized hotspots where thermal runaway may be initiated, in a heavily degraded cell [8]. Imbalanced cells may experience voltage spikes or current surges during charging or discharging cycles, especially when transitioning between different operational states. These irregularities in voltage and current can induce stress on the cells and lead to thermal runaway if not properly managed by the battery management system [5].

The sum of these factors can start, or contribute to, an incident of thermal runaway. As the lithium-ion cells' temperature increases, the chemical reactions in the cell get faster and faster, leading to a self-feeding reaction. This catastrophic effect of self-heating, known as thermal runaway, causes the expansion of the cells, production of smoke, fire, and finally, leads to the explosion of the cell pack. If BMS detects the development of this effect too late and a battery enters a state of thermal runaway, it usually cannot be stopped. Outside thermal influence from cooling mechanisms, and an attempt to prevent this only before the thermal runaway has been initiated [5].

2.5.1 Passive Cell Balancing

Because of these issues and the critical flaw of lithium-ion packs, a BMS must regularly balance the battery cells and maintain a certain degree of uniformity for the SOCs. There are many methods for ensuring this balance, each requiring different circuitry and electronics within the pack to enable it, and they are categorized into two categories: passive and active cell balancing [8] [26]. In terms of energy, passive balancing usually is classified as dissipative, whereas active balancing is non-dissipative [27].

Passive cell balancing is the simplest form of balancing, where cells are discharged until they reach similar SOCs. The excess charge is transferred into heat through a consumer, a resistor. The main benefits of these methods are simple implementations and relatively cheap components. There is no need for advanced algorithms. In some implementations, the BMS has no direct influence on the passive balancing mechanism, which functions purely at the circuit level [8].

The downside of these methods is that battery charge is wasted destructively. Energy may not be simply removed from the cells and is instead transformed directly into heat. The resistors are placed close to the cells and this leads to the heating of the pack. Due to this effect, as well as the inaccuracy of SOC estimations during charge/discharge, it can be dangerous to allow this balancing to happen outside resting periods, where there is no outside current applied to the cells. Applying this method during charge/discharge comes with the danger of overheating the battery which, even if thermal runaway is avoided, still contributes to the deterioration of the cells [8] [26].

2.5.2 Active Cell Balancing

Active balancing, as a non-dissipative type does not waste cells' energy to achieve balance. Most active cell balancing implementations rely on transferring energy from high SOC cells to low SOC cells. There is some waste of energy on the circuitry needed to enable the energy transferring between cells, but lower in comparison to the passive balancing strategies [8].

Naturally, transferring energy between different cells requires more complex mechanisms, some of which are not effective enough to be considered as good alternatives to already implemented passive balancing designs. Not only in complexity, but voltages between imbalanced cells need to be relatively large so that cells can balance quickly. This means that low differences in cell voltages, even though they can be unbalanced, can make the energy transfer too slow, unable to achieve balance in the necessary amount of time without the need for additional circuitry that can bypass this limitation.

2.6 Cell Model

There are a variety of different models utilized to simulate battery cell behaviors, some of which are Electro-chemical Models (EM), Equivalent-circuit Models (ECM), and Data-driven Models (DDM) [28], each with different advantages and disadvantages.

In this project, we utilize the ECM model for cell and cell pack simulation.

2.6.1 Equivalent-circuit Model (ECM)

ECM models the electrical behavior of battery cells through circuit theory, utilizing electrical elements such as resistors, capacitors, inductors, and voltage/current sources. The ECM model used is based on the Thevenin circuit model, one of the most widely used ECM models for cell battery simulation, given that it can more accurately represent the dynamic behaviors of the cells [28]. Due to the Thevenin ECM model being a linear and time-invariant circuit model, it does not capture accurately the nonlinear and time-variant physical behaviors of the cells. Additionally, for the model to be accurate, real cells must be measured and parameterized to tune the model parameters for precise simulation. This can be challenging given the non-linear behaviors that cells show depending on different parameters such as temperature, SOC, and rates of charge and discharge [8], [28].

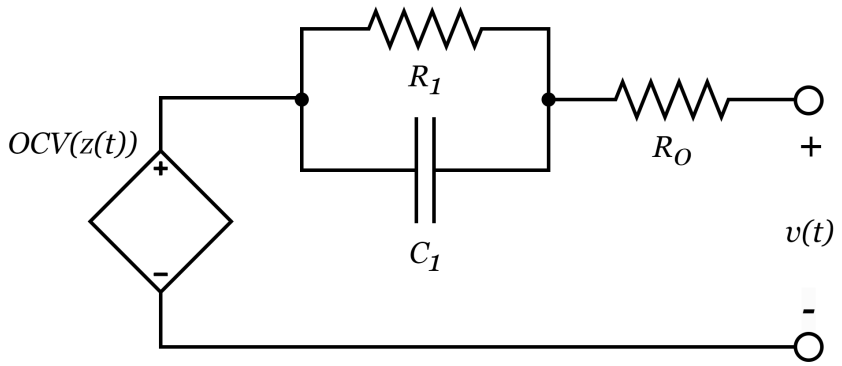


Figure 2.3: First order Thevenin model of a cell.

In this model, OCV is defined as the Open Circuit Voltage, which is the voltage of the cell measured without load. The OCV is also a necessary parameter to determine the SOC of a given cell. The cell model voltage is modeled as shown in equation (2.3).

$$v(t) = OCV(z(t)) - R_1 i_{R_1}(t) - R_0 i(t) \tag{2.3}$$

2.7 Reinforcement Learning

Reinforcement learning is a branch of machine learning. Machine learning, as a concept, is a computational approach to learning. Reinforcement learning uses 'interaction' as its object of learning. The 'learner' in an RL setting is frequently called the 'agent'. The objective of the agent is to solve an infeasible problem to approach through traditional algorithmic means [29]. The usual applications of RL algorithms are problems with a, often large, set of variables in a non-uniform environment. That is to say, they influence each other non-linearly [29].

Reinforcement learning is formalized as control optimization in an incompletely-known Markov decision process. The learning agent must sense the state of an environment and take actions to affect that state. RL is often referred to as a form of unsupervised learning, however, the objectives of these two methods are different. Unsupervised learning tries to find a structure hidden in collections of unlabeled data, whereas reinforcement learning tries to maximize a reward sequence rather than trying to find a hidden structure. Although there is some overlap in these objectives, as finding a structure would be beneficial for RL in most situations, however, it does not guarantee a maximization of a reward [29].

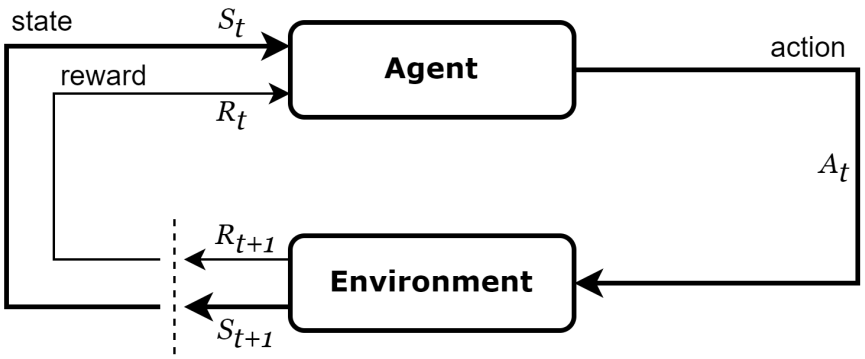


Figure 2.4: RL training principles

Figure 2.4 illustrates the the dataflow of RL training. An environment (i.e. the battery back) sends out states to the RL agent. The agent provides the best course of action or that state that it has learned through experience so far. The action alters the environment, leading to a new state. This state is assigned a reward, according to the reward function, and the feedback is given to the RL agent. After enough state-transition reward pairs are gathered, the RL agent goes through a training step in which it updates its course of action.

An RL agent's target for interaction is the environment. An environment can be defined as the setting in which our agent can act and influence elements within. It can be several integer variables, a set of continuous arrays, or a complex three-dimensional simulation. The definition of an environment depends on the problem which we seek to solve. The RL agent is offered an interface through which it can interact with the environment. This is the Action Space of our environment, and it represents variables that are manipulable by our agent. The variables that cannot be directly influenced by our agent are termed the Observation Space. A 'snapshot' of the Observation Space at a certain timestep is called a 'state'. States are linked together by the actions which the agent takes between them. We start interaction with an environment in an initial state, and each action the agent takes ends in another state. Actions are sequential and begin from the resulting state of the previous action [29].

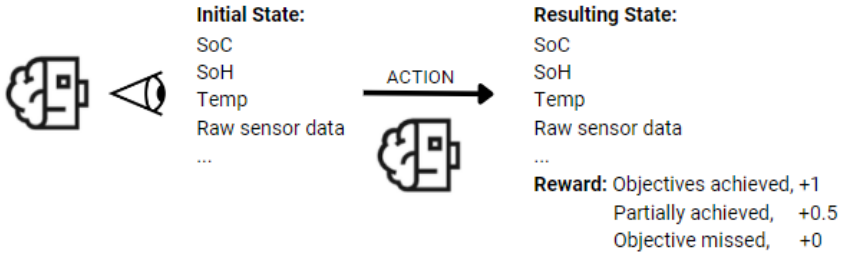


Figure 2.5: RL agent state-action-reward unit

Each action taken upon an initial state and leading to a final state is assigned a reward, formalized in Equation (2.4). The reward is defined by a reward function that represents

the target of our RL agent, the objective which it must achieve through its actions, bounded by a discount factor γ which takes values between 0 and 0.99 Equations (2.5, 2.6). The value discount factor determines how much the RL agent 'looks ahead' as it tries to maximize the rewards. The largest instance of the sum is the immediate reward for a state-action pair, however, with a large discount factor, the future rewards for the following actions gain more importance. An agent may thus take an action that has a low immediate reward but allows for much higher rewards for future states. The agent then develops efficient long-term strategies that can secure high total rewards by taking into consideration the development of states. An RL agent with a discount factor of 0 is known as 'myopic', and ignores future considerations in its decision-making. It has a maximum possible value of 0.99, which is a mathematical technique to ensure rewards in the distant future eventually approach 0. This ensures the immediate action reward has more importance than individual future actions. The predictive focus of an RL algorithm must be tuned to the specific task it aims to solve, and the discount factor is critical in defining this focus [29].

$$r_t = R(s_t, a_t) \tag{2.4}$$

The RL objective is to select a policy that maximizes the expected reward sum when the agent acts according to that policy. A policy is a stochastic rule by which the agent selects actions as a function of states. A policy's value functions (v_π and q_π) assign to each state, or state–action pair, the expected return from that state–action pair, given that the agent uses the policy (2.5). The value function v of a state s under a policy π, denoted $v_\pi(s)$, is the expected return when starting in s and following π thereafter. Similarly, we define the value of taking action an in state s under a policy π, denoted $q_\pi(s, a)$, as the expected return starting from s, taking the action a, and thereafter following policy π (2.6) [29] [16].

$$v_\pi(s) = E_\pi[G_t|S_t = s] = E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s\right] \tag{2.5}$$

$$q_\pi(s, a) = E_\pi[G_t|S_t = s, A_t = a] = E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a\right] \tag{2.6}$$

These value functions are estimated through experience, an agent follows the policy π and computes an average for the states encountered of the total reward returns that have followed that state, which converges to the state's value $v_\pi(s)$ as the number of times that state is encountered nears infinity. Keeping a separate average for each action in each state, we converge to the action values $q_\pi(s, a)$. The value functions define ordering over policies. A policy π is defined as better than π' if the expected total reward is greater than that of π' in all states. Formally, $\pi > \pi'$ if and only if $v_\pi(s) > v_{\pi'}(s)$. The RL task is to find the optimal policy π_* which is greater than all other policies, formalized in the following equations (2.7), (2.8), (2.9) [29].

$$v_*(s) = \max_\pi q_\pi(s) \tag{2.7}$$

$$q_*(s,a) = \max_{\pi} q_\pi(s,a) \tag{2.8}$$

$$q_*(s,a) = E[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a] \tag{2.9}$$

Traditional greedy algorithms in non-linear environments show severely limited performance. A greedy algorithm begins by sampling an action, and will then continuously take that action as it is the 'most rewarding' action at its disposal. It makes no effort to explore the other actions it can take, as doing so is not according to its imperative of maximum immediate reward, which it can only compute from the actions it has taken. Thus, the same course of action is repeated *ad infinitum* with no improvements, never discovering the better alternatives offered by the other actions. RL is defined by its capacity for effective exploration of a large environment [29].

A big challenge in designing RL systems is the balance between exploration and exploitation. To accrue a large reward, the agent must take actions that it has tried in the past and knows will produce an effective reward. However, to discover such actions the agent must try actions that it did not attempt before. Our agent must exploit what it has already tried, and explore to discover new methods for achieving rewards. By choosing one, we fail the other, and a balance must be struck. This trade-off dilemma is specific to reinforcement learning and has no equivalent in supervised or unsupervised learning [29] [30] [31].

RL algorithms utilize techniques such as e-greedy, softmax, or Upper Confidence Bound (UCB) to encourage the agent to explore a wide range of actions and states in the early stages of learning. This exploration helps the agent discover effective strategies and solutions that might not be immediately apparent. By systematically balancing exploration with exploitation, RL ensures that the agent can both discover new strategies and refine them to maximize long-term rewards, eventually discovering a final optimal policy [29].

The biggest bottleneck in RL is samples. RL algorithms need many state-action-reward samples to train efficiently. The number of samples varies by task/environment but usually reaches several million state transitions for convergence. An RL algorithm converges when further exploitation yields no further improvements, and the exploration mechanism tapers off. For example, some algorithms utilize an entropy parameter that gradually decreases according to loss functions based on the accuracy of predicted rewards, while others introduce noise to the action of the agent to ensure exploration [15] [16].

There are two approaches to RL when it comes to sample-gathering: Online RL and Offline RL.

In offline RL, samples are provided from interactions with other actors (i.e. other BMS control algorithms). These interactions are recorded from logged data and tagged with rewards. In the training process, the RL agent cannot directly act upon the environment and instead learns from the actions of other controllers.

In online RL, an environment is simulated and the RL agent interacts directly with the simulation to generate state-action-reward samples. This approach works best when

there is no abundant amount of usable logged data to train the RL agent with. It converts the sample scarcity problem into a computation-power problem. However, the simulation must be relatively accurate to the real-world environment we seek to learn [29].

Examples of RL infrastructures in the industry initially start with an online approach when branching into the area, and gradually move to Offline once they have the possibility of generating enough logged data from products, or through digital twins. A digital twin is a simulation that we can say is certifiably identical to its real-world counterpart. Online RL may also offer more possibilities for exploration than Offline learning from traditional controllers. However, the accuracy of the simulations remains a key issue.

As we are experimenting with a novel topology for our approach, there is no widely available large bank of logged BMS interaction data. Within our work, we rely on simulations to generate the necessary samples for the training process, via direct interaction from the RL agent in an online RL setup.

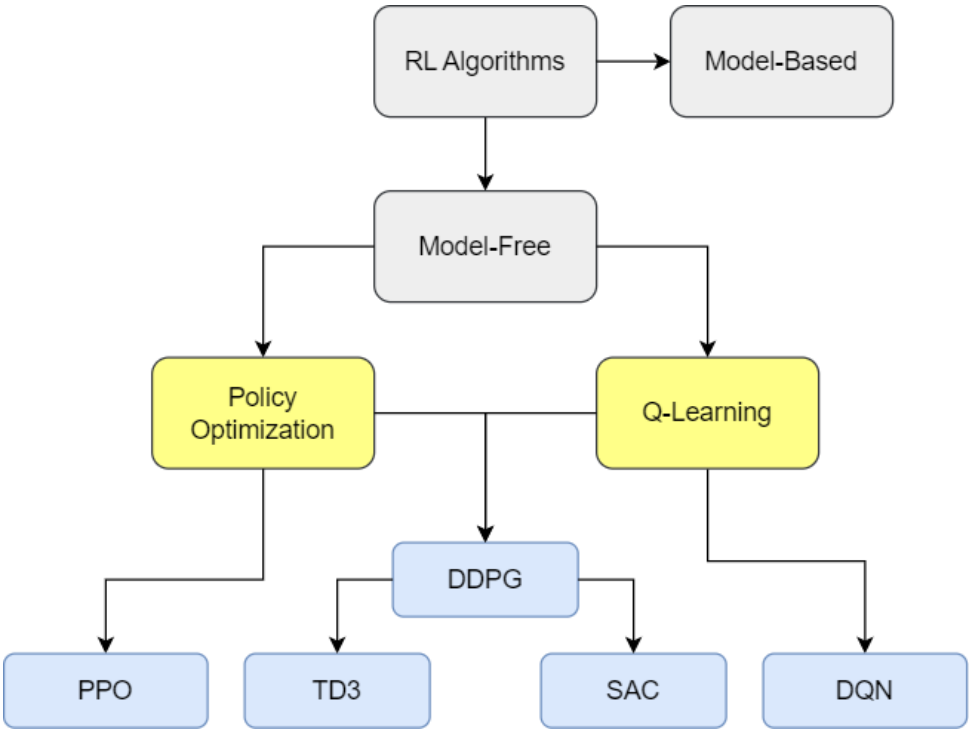


Figure 2.6: Taxonomy of RL algorithms

Reinforcement learning algorithms are classified into two large categories: model-based and model-free.

Model-based algorithms learn an explicit model of the environment, in the form of transition probabilities or a predictive model. They utilize this model to plan and make decisions about actions to take. That is to say, a model-based approach implies the possibility of predicting rewards and states, during training, before an action is taken. Rather than relying on trial and error, the model-based approach uses approximations of the environment to view the possible outcomes before taking a step [32].

Model-based RL approaches require certainty that the approximation fully captures the essence of the environment, one that is identical to the ground truth. It is also crucial for this approximation to be very efficient in generating predictions, otherwise the time-to-train becomes infeasible. These approaches work best with easily modeled environments with few or no non-linear factors, such as games of Chess and Go [32].

Model-free approaches are applied directly to an environment, and learn through trial and error. A model-free agent must take an action, and see the results, to be able to learn from it. This grants much more flexibility and eliminates the risks of inaccurate models training unusable agents. It comes with the cost of less sample efficiency, as we need to run more simulations for the agent to find the best course of action [32].

Within the model-free methods, we have several classifications: Policy Optimization, and Q-learning methods.

Q-learning algorithms are also known as 'off-policy'. They learn a value function, which is then used to derive a policy. Specifically, it learns the action-value function (Q-function), which predicts the expected utility of taking a given action in a given state and following a certain policy thereafter. It does so by estimating the Bellman equation. A big advantage of off-policy methods is that they can use data collected at any point during training, even from past actions from previous, less-trained, versions of the agent. The most popular algorithm representative of this class, which spawned this branch of RL, is Deep Q-Network (DQN) [33].

Policy optimization, or 'on-policy' methods can only use data collected with the latest version of their policy. They learn a policy function that directly maps states to actions. The approach of policy-based methods makes them very stable and able to handle continuous action spaces, and smoothly converge to an optimal policy [18]. On-policy algorithms are ideal for scenarios with a large and continuous action space. They do however suffer from inefficiency in sample usage.

3

Methods

3.1 Balancing Strategy

As discussed previously, balancing can be categorized as dissipative and non-dissipative. For this project, we propose a non-dissipative balancing strategy that does not rely on transferring charge between cells. We explore the idea of each cell being discharged at different rates, by the normal operational power-draw of the vehicle, in order to achieve balance, for this to occur cells must be balanced during discharge phases, namely, during operation.

Balancing during operation can allow the cell pack to be utilized fully, by allowing higher SOC or stronger cells to instead take part of the work of the low SOC or weak cells. With this, low SOC or weak cells get used less, balancing the overall power demand between all cells equitably so that cell aging averages out in a similar manner on all cells, rather that having cells aging more quickly that other. Ideally, when cell packs reach the end of their lifetime, all cells should have been fully exploited.

To allow for balancing during operation, we utilize a cell pack topology which is a re-configurable battery topology, illustrated in Figure 3.1. In this battery pack, each cell is connected to a power electronics circuit unit whose operation is to modulate the amount of charge that each individual cell will provide. Each circuit unit, for the purposes of this project, is connected in series with other units, forming a pack. By being able to modulate the amount of charge each cell provides, it is possible to remove the needed charge of each cell during operation with the aim of achieving balance.

In Figure 3.1, the buck-boost converter represents the power electronics section of the electronics while the controlling switch represents the circuit that will handle the charge flow accordingly. This is a representation of what could be expected of the controller power electronics to be, however the electrical design and considerations of this are out of scope for this work. Any similar circuit or system can be used instead of the presented one, as long as the handling circuit is capable of executing the demanded tasks implied by this work.

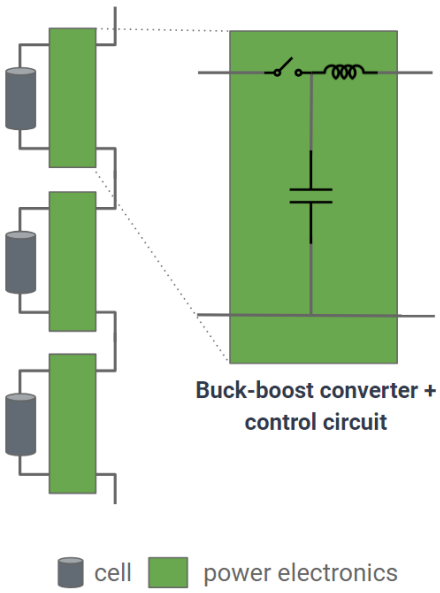


Figure 3.1: Balancing focused re-configurable cell pack topology.

3.2 Battery Simulation Environment

This section provides details about the cell simulation, the data used for the simulations, the interfacing between the simulation environment and the machine learning training environment and how balancing is handled through the simulation.

3.2.1 Cell Simulation

The cell simulation code was leveraged from previous works [8] and in order for these to fit the objectives of this project, the code was modified and new code was added onto it.

The cell simulation reproduces a cell's electrical parameters over time based on the demanded power profile, utilizing the electrical cell model described in section 2.6 Cell Model. Every cell simulation uses the same cell model and parameters, as shown in Table 3.1, which correspond to cell model "P14" when running the cell simulation environment. More cell models are available for this work, however only cell model "P14" was used for the final experimental results.

Table 3.1: "P14" cell model parameters.

Parameter	Value	Unit	Description
T	25	Degrees Celcius	Temperature of operation of the cell
q	14.53	Amp × hour (Ah)	Cell capacity
eta	0.999	Ratio, unitless	Coulombic efficiency
r0	0.00178096	Ohms	Series resistor parameter
r	6.477208e-04	Ohms	Resistor-Capacitor resistance
rc	0.823683	Ohms	R-C resistance
rt	2.5e-4	Ohms	Cell tabs resistance

The temperature parameter for this work has been set to 25 °C. Most of the parameters of the cells change depending on the temperature in varying degrees but mostly stay relatively similar. Given that this would greatly increase the complexity and time of the simulation, the temperature will remain constant for the purposes of this work.

Each cell is simulated in a pack of 10 cells which have slight differences in their parameters, in order to simulate factory irregularities, aging differences, and capture the unbalancing characteristics of the cells. If the cells in the simulation have the exact same parameters, then unbalancing will never occur since they will all act exactly the same way. To avoid this, some cell parameters have small variations. This is meant to emulate real cells, given that the manufacturing process of the cells cannot possibly yield virtually exact cells, thus this parameter variance is reasonable to have. To have some control over the randomness of the parameter values, each random parameter is seeded via a seed parameter that can be freely chosen on each simulation - this allows simulations to be accurately reproduced as well as being able to have reproducible yet random cell configuration.

The parameters $q, eta, r0$ for cell i are randomized as follows

$$Q_i = q - 0.25 + 0.5\alpha_i \tag{3.1}$$

$$Eta_i = eta - 0.002\beta_i \tag{3.2}$$

$$R0_i = r0 - 0.0005 + 0.0015\gamma_i \tag{3.3}$$

where $\alpha_i, \beta_i, \gamma_i \sim \text{Unif}\,(0,\, 1)$. Here $\alpha_i, \beta_i, \gamma_i$ and $\alpha_j, \beta_j, \gamma_j$ are independent when $i \neq j$.

During simulation, different voltages and currents are calculated depending on the power that is being demanded on each cell, and ultimately the SOC of each cell is computed. SOC is one of the primary values that is used for balancing, it also is used for the simulation itself, given that cells start with a set initial SOC value from which the rest of the parameters are obtained.

The open circuit voltage (OCV) v_{OC} is also part of the cell model parameters. The cell open circuit voltage v_{OC} is a function that depends on the cell SOC and temperature, defined as follows

$$v_{\text{OC}} = OCV fromSOC(SOC, T, model) \tag{3.4}$$

where SOC is the current state of charge of the cell, T is temperature and $model$ are the "P14" cell model parameters and $OCV fromSOC()$ is a lookup table function that finds the cell's v_{OC} depending on the input parameters. The cell $model$ contains all the parameters shown in Table 3.1 as well as the voltage-SOC relationship of the simulated cell. This data belongs to the cell parameters we utilize and was obtained through cell characterization tests and measurements, which we leverage in this work.

Figure 3.2 illustrates the relationship between v_{OC} and SOC, where SOC is the free variable.

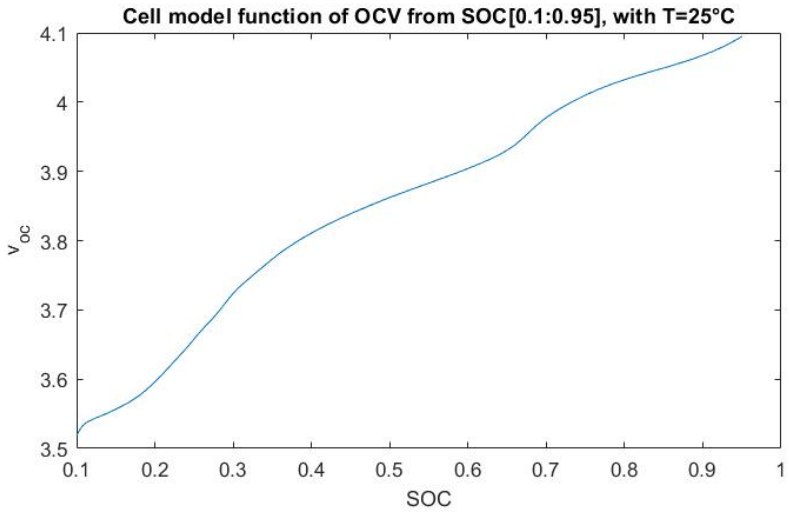


Figure 3.2: "P14" cell model of OCV and SOC relationship, T=25 C°.

The initial value of SOC for all cells is the same, this is the free variable of the simulation which needs to be set at initialization. It is set to the upper SOC limit at the start of the simulation:

$$SOC(0) = maxSOC \tag{3.5}$$

Later, once the simulation has calculated the voltages and currents going through the cells, the SOC is calculated again and updated, from which the simulation cycle begins again.

By defining the maximum and minimum limits for SOC, $maxSOC$ and $minSOC$ respectively, we can use Equation (3.4) to calculate the respective maximum and minimum voltage limits that the cell can have, $maxVlim$ and $minVlim$ respectively - shown in Table 3.2.

The simulation parameters that are set at initialization are shown in Table 3.2.

Table 3.2: Cell simulation initial parameters.

Parameter	Value	Unit	Description
maxSOC	0.95	Percentage	Upper SOC limit for cell
minSOC	0.10	Percentage	Lower SOC limit for the cell
maxVlim	4.095	Volts	Upper voltage limit for the cell
minVlim	3.5185	Volts	Lower voltage limit for the cell
leak_c	0.01	Ampere	Leakage current of the cell
Tsd	20	Celsius	Self-discharge cell temperature

The values for $maxSOC$, $maxSOC$, $leak\_c$ and Tsd are simulation defined, while $maxVlim$ and $minVlim$ are values obtained from the OCV-SOC relationship from the cell model, via Equation (3.4).

Leakage current, which is parameter $leak\_c$, aims to simulate cells slowly losing charge over time due to an external factor. This is to mimic cells becoming discharged due to circuitry utilizing cell charge for computation or control circuitry, such as powering the BMS control circuit.

Additionally, cells can exhibit a self-discharge behavior, the simulation treats the self-discharge of the cell as an additional current. This is a different current than the leakage current, and the parameter that controls this current is Tsd - self-discharge cell temperature.

The parameters $leak\_c$ and Tsd for cell i are randomized as follows

$$Leak\_c_i = leak\_c + 0.002\alpha_i \tag{3.6}$$

$$Tsd_i = Tsd + 10\beta_i \tag{3.7}$$

where $\alpha_i, \beta_i \sim \text{Unif}\,(0,\, 1)$. Here α_i, β_i and α_j, β_j are independent when $i \neq j$.

Once the initial parameters are defined, the simulation can start calculating the several variables that are dependent on the discharge power profile used. Cells for passive balancing are considered connected in series and simulated as such, while cells for reconfigurable active balancing are simulated individually, with no direct connection to other cells.

For passive and active balancing, the current i_{net} represents the current flowing through any cell and is defined as follows

$$i_{net}(t) = i_{app}(t) + i_{self\text{-}discharge}(t) + i_{leakage}(t) \tag{3.8}$$

Where $i_{app}(t)$ is the discharge current, dependent on the power profile, $i_{self\text{-}discharge}(t)$ is the self-discharge current and $i_{leakage}(t)$ is the leakage current.

The discharge current i_{app} is initialized as follows

$$i_{app}(0) = 0 \tag{3.9}$$

Cell pack simulation for passive balancing topology:

$$\sum_{j=1}^{N} \left(v_{\mathsf{OC},j} - v_{\mathsf{RC},j}\right) i - \sum_{j=1}^{N} R_{0,j} i^2 = P \times N \tag{3.10}$$

where N is the number of cells in a battery pack, i_{app} is the current through the pack, and P is the power usage of a single cell, read from the drive cycle profile. For cell j, $v_{\mathsf{OC},j}$ is the open circuit voltage, $v_{\mathsf{RC},j}$ is the RC circuit voltage, and $R_{0,j}$ is the resistance in series.

To calculate i_{app} with all voltages positive, we use

$$i_{app} = \frac{\sum_{j=1}^{N} \left(v_{\mathsf{OC},j} - v_{\mathsf{RC},j}\right) - \sqrt{\left(\sum_{j=1}^{N} \left(v_{\mathsf{OC},j} - v_{\mathsf{RC},j}\right)\right)^2 - 4(P \times N) \sum_{j=1}^{N} R_{0,j}}}{2 \sum_{j=1}^{N} R_{0,j}}. \tag{3.11}$$

Cell pack simulation for reconfigurable active balancing topology:

For cell j, $j \in [1, 2, \cdots, N]$

$$\left(v_{\mathsf{OC},j} - v_{\mathsf{RC},j}\right) i_j - R_{0,j} i_j^2 = P \tag{3.12}$$

where i_{app_j} is the discharging current, P is the power usage of a single cell, obtained from the drive cycle profile, $v_{\mathsf{OC},j}$ is the open circuit voltage, $v_{\mathsf{RC},j}$ is the RC circuit voltage, and $R_{0,j}$ is the resistance in series. To calculate i_{app_j} with all voltages positive, we use

$$i_{app_j} = \frac{\left(v_{\mathsf{OC},j} - v_{\mathsf{RC},j}\right) - \sqrt{\left(\left(v_{\mathsf{OC},j} - v_{\mathsf{RC},j}\right)\right)^2 - 4PR_{0,j}}}{2R_{0,j}} \tag{3.13}$$

The current calculation for $i_{self\text{-}discharge}(t)$ for cell j is defined as

$$i_{self\text{-}discharge}(t) = \frac{\left(v_{\mathsf{OC},j} - v_{\mathsf{RC},j}\right)}{((-20 + 0.4Tsd) \times SOC_j + (35 - 0.5Tsd)) \times 1000} \tag{3.14}$$

As for leakage current $i_{leakage}$ for cell j the value remains constant as shown in Formula (3.6) and is defined as

$$i_{leakage_j} = Leak\_c_j \tag{3.15}$$

The RC voltage $v_{\mathsf{RC}}(t)$ of any cell i is the voltage of the RC components of the ECM model, which is necessary for calculating the terminal voltage $v_{\mathsf{t}}(t)$ of the cell. The RC voltage $v_{\mathsf{RC}}(t)$ the can be calculated as follows

$$v_{\mathrm{RC}i}(t) = r_i \times i_{RC,i}(t) \tag{3.16}$$

Where r_i is the RC resistance, from Table 3.1, and $i_{RC,i}(t)$ is the current of the RC components of the cell model. The RC current $i_{RC}(t)$ of cell i is defined by the following

$$i_{RC,i}(0) = 0 \tag{3.17}$$

$$i_{RC,i}(t+1) = rc_i \times i_{RC,i}(t) + (1 - rc_i) \times i_{neti}(t) \tag{3.18}$$

Where rc_i is the RC resistance value (cell parameter from Table 3.1) and $i_{neti}(t)$ is the current flowing through the cell.

With the cell OCV voltage v_{OC} and the RC voltage v_{RC}, the terminal voltage v_{t} of cell i can be calculated as follows

$$v_{\mathrm{t}i}(t) = v_{\mathrm{OC}i}(t) - v_{\mathrm{RC}i}(t) - i_{neti}(t) \times r0_i \tag{3.19}$$

Where $v_{\mathrm{t}i}(t)$ is the terminal voltage, $v_{\mathrm{OC}i}(t)$ is the open circuit voltage, $v_{\mathrm{RC}i}(t)$ is the RC voltage, $(i_{neti}(t)$ is the current flowing through the cell and $r0_i$ is the $r0$ resistance parameter of the cell.

Finally, the SOC of the cell can be calculated, which will be the new SOC value for the next simulation iteration. The SOC for cell i is calculated as follows

$$SOC_i(t+1) = SOC_i(t) - \frac{1}{3600} \times \frac{i_{neti}(t)}{q_i} \tag{3.20}$$

Where $i_{neti}(t)$ is the current flowing through the cell, and q_i is the cell's capacity.

Additionally, in this work we utilized state of health (SOH) as the secondary parameter for balancing in addition to SOC. For SOH, the power expended by every cell is captured and accumulated. The calculation for SOH of cell i is defined as follows

$$SOH_i(0) = |(v_{\mathrm{OC}i}(0) - v_{\mathrm{RC}i}(0)) \times i_{neti}(0)| \tag{3.21}$$

$$SOH_i(t) = SOH_i(t-1) + |(v_{\mathrm{OC}i}(t) - v_{\mathrm{RC}i}(t)) \times i_{neti}(t)| \tag{3.22}$$

3.2.2 Drive Cycle Profiles

The demanded power for each cell refers to the expected power output each cell must be providing at any given moment. This power value is converted by the simulation to the current that must go through the cells in order to supply the demanded power, this is calculated using the dynamic variables such as voltage as well as the internal cell parameters. With the power demand, the current that is expected to go through each cell is also calculated. The current going through each cell will either charge or discharge the cell, depending on which state of operation cells are in.

Since the cell simulation requires for a power demand profile that will allow cells to provide energy and become discharged over time, the simulation utilizes drive cycle profiles based on real-world data, leveraged from previous works. This allows for cells to be discharged at a rate which is the closest to a real-world scenario as possible, so that we can capture more realistic cell discharge behaviors.

The drive cycle profiles are a function of expected power demand versus time. The Power unit is in Watts and time is measured in seconds in the simulation. There are multiple drive cycle profiles of an EV's power demand over time in a given driving environment, such as in a city, highway or urban area. Each of these scenarios have highly different characteristics in terms of power demand, for instance, on a highway the power demand is the highest given the need to maintain a high speed is energy-intensive, contrary to driving in a slow area with multiple stopping points in an urban area.

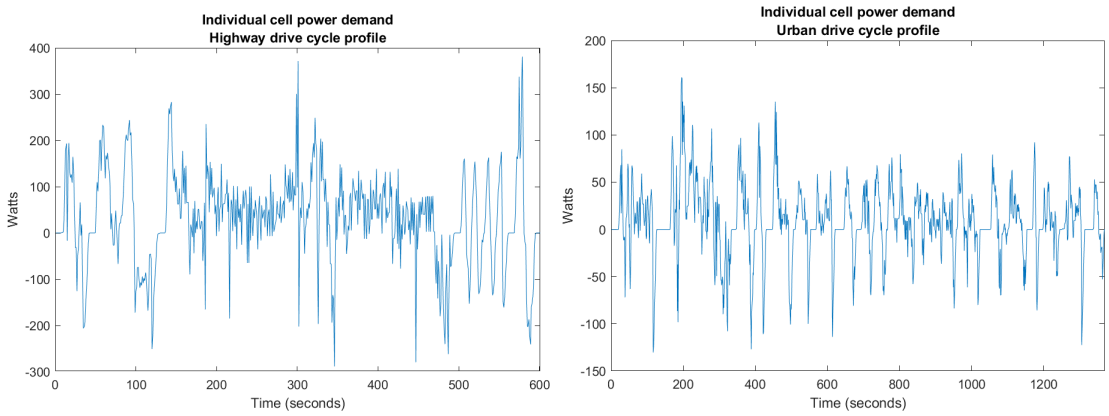


Figure 3.3: Highway (us06) and urban (udds) drive cycle profiles.

Figure 3.3 is a graph that shows the power demand profiles for one cell. The drive cycle of a highway profile is "us06" and the urban drive cycle profile is "udds", leveraged from [34]. These are the two main used profiles in the simulation experimentation. The "us06" profile discharges cells at a much higher rate and has a short duration, this means that cells under constant discharge with this profile will require to be charged more often. On the other hand, the profile "udds" has a lower power demand and longer duration, this translates to cells becoming discharged much slower and requiring charging less often.

3.2.3 Cell Simulation States

The cell simulation accounts for three different states where cells could be in: charging, discharging, and resting. These three different states aim to imitate the normal operation of cells on a real system. Cells will become discharged during usage, charged when they reach a certain lower threshold, and resting when the system is not under operation. Figure 3.4 shows the state-flow of the cell simulation.

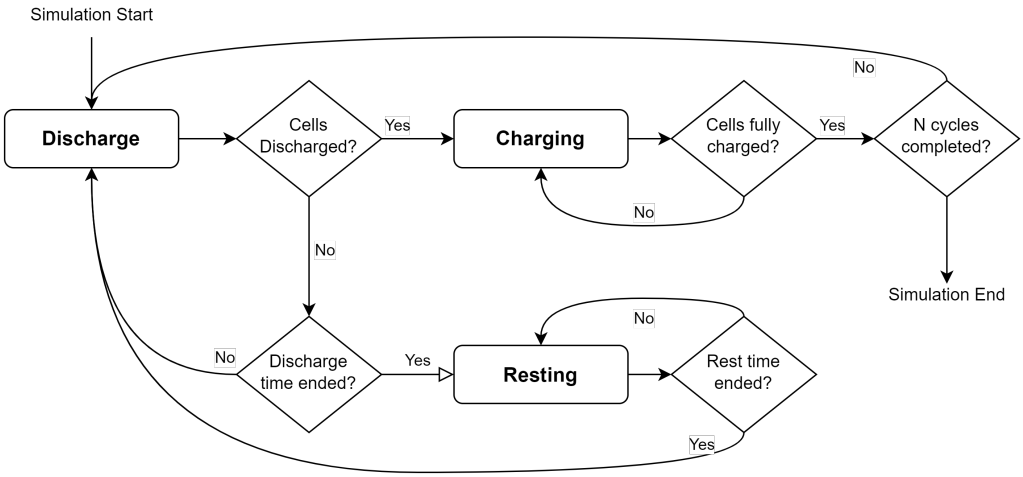


Figure 3.4: State flow chart of the cell pack simulation.

The simulation time scale depends on the power profiles, and each simulation step simulates one second of operation. Each step, SOC along with other variables are calculated.

During "discharging" phase, cells become discharged based on the demanded power profile as well as leakage and self-discharge currents. At every time step, SOC is calculated for each cell, and later cell SOCs and cell voltages are checked, if they are below a certain threshold, then the simulation state changes to "Charging".

Any cell is i considered discharged when

$$v_{t,i}(t) <= minVlim \text{ OR } SOC_i <= minSOC \tag{3.23}$$

Where $v_{t,i}(t)$ is the terminal voltage of the cell and SOC_i is the SOC of the cell. If any of these conditions are true, the simulation state changes to "charging".

In order to control how long cells are under operation and rest, two counter variables are used. The internal parameter $usageCounter$ controls the time cells will be charging and discharging, and the internal parameter $restingCounter$ controls how long cells will rest. After $usageCounter$ finishes, $restingCounter$ starts. Once $restingCounter$ ends, this is considered a cycle. The amount of cycles the simulation will run for is provided by the user during the set-up and configuration of the simulation environment (Section 3.7), illustrated in Figure 3.4 as N. Once the simulation has executed N cycles, it ends.

For the "charging" state, cells are charged at a rate of $6.6KW$. Once any cell in the pack has reached its maximum voltage or SOC threshold, charging stops and the simulation returns to its discharging state.

Cells during charge are not capable of fully receiving all the charge due to inefficiencies with energy transfer. The parameter that defines the efficiency transfer is Coulombic Efficiency, or eta (η) as defined in Table 3.1. This parameter affects the charging current directly.

When charging, the current $i_{net}(t)$ flowing through the cells is defined as

$$i_{net}(t) = i_{app}(t) \times eta + i_{self\text{-}discharge}(t) + i_{leakage} \tag{3.24}$$

where $i_{app}(t)$ is the charging current, $i_{self\text{-}discharge}(t)$ is the self discharge current and $i_{leakage}(t)$ is the leakage current.

Any cell is i considered charged when

$$v(t)_i => maxVlim \tag{3.25}$$

where $v(t)_i$ is the terminal voltage of the cell.

The simulation will reach the "resting" state after $usageCounter$ ends, and will remain in this until $restingCounter$ ends. Essentially, after a certain amount of time that the simulation is "discharging", it will change to "resting" to emulate cells under no load. During this state, cells will only be subject to leakage and self-discharge currents. If cells are left resting infinitely, they will eventually fully discharge.

During the resting state, cells are not being charged nor discharged, this means that the only current flowing through the cells is leakage and self-discharge currents.

As such, any cell is considered in rest when the current $i_{app}(t)$ is zero. Where $i_{net}(t)$ is the current of the cell, resting current is defined as follows from Equation (3.8)

$$i_{app}(t) = 0. \tag{3.26}$$

Thus

$$i_{net}(t) = 0 + i_{self\text{-}discharge}(t) + i_{leakage} \tag{3.27}$$

$$i_{net}(t) = i_{self\text{-}discharge}(t) + i_{leakage} \tag{3.28}$$

Where $i_{net}(t)$ is the total current flowing through any cell during rest.

Figure 3.5 shows how the SOCs of different cells look like over time during several simulation cycles and phases, as well as a plot of the difference of maximum SOC against minimum SOC - this is meant to illustrate how cells tend to unbalance over time. Note that the SOC differences increasingly grow over time, when there is no balancing policy controlling the simulation.

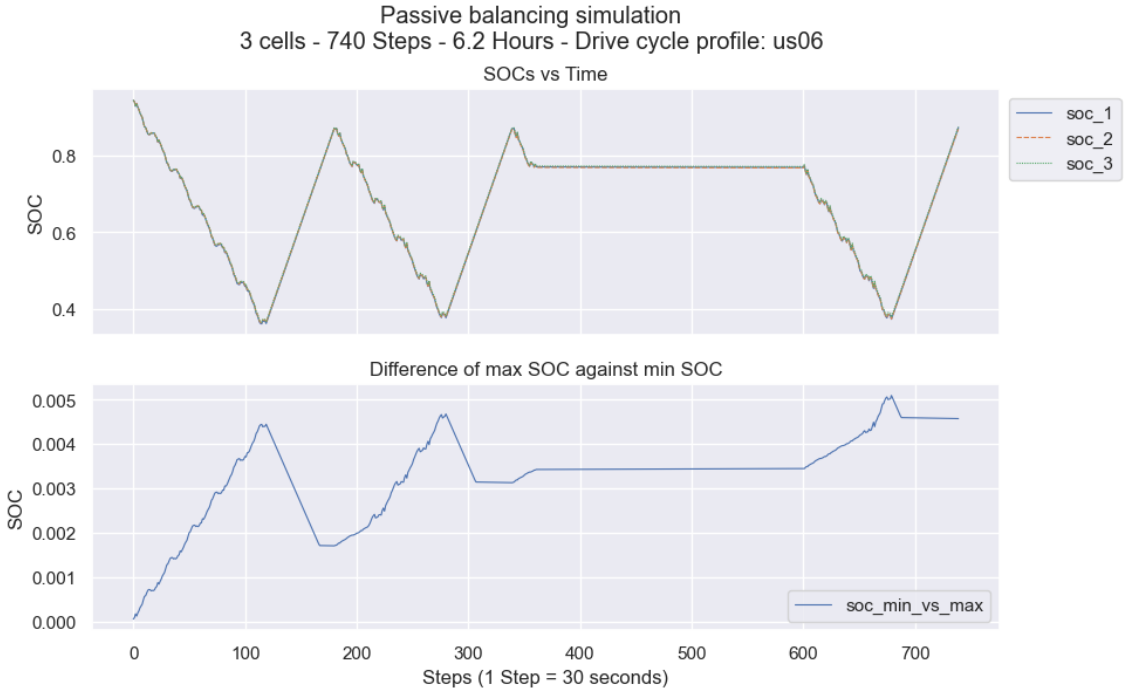


Figure 3.5: Cell pack SOCs and SOC difference during discharging, charging, and resting phases.

3.2.4 Cell Balancing Simulation

Balancing in the simulation is handled for both passive and reconfigurable active balancing. For this to work along with the RL model training environment, the simulation has a feedback loop from where it obtains the actions that the RL model wants to take and applies them to the simulation, directly affecting the simulation calculation when balancing. This means that the simulation cannot continue unless it has received actions from the RL model, given that this is an online training methodology.

Due to resource limitations when executing the simulation environment and passing information to the RL training environment, data passing had to be down-sampled between processes. Sending each simulation step to the RL training environment is a very slow process given that it involves transferring information to another process and waiting for that other process to compute the next action and return the action to the simulation. As a result, the simulation has a down-sampling factor of 30, which drastically reduced simulation and training time. As a consequence of this, the actions applied when balancing persist for 30 simulation steps, until the next value is received and applied for balancing.

Given that active and passive balancing occur at different stages of cell operation, the balancing strategy and feedback will be different for each. For passive balancing, cells need to be discharged over time while they are in the resting phase only. For the active balancing approach presented in this project, balancing occurs during cell operation only. Any phases where cells cannot be balanced, they will become naturally unbalanced if no balancing strategy is applied.

Passive balancing feedback for the simulation is treated as a discharging current that flows through each cell individually. Each cell will become discharged based on the balancing strategy of the RL model. The RL balancing model has full control on the cell current discharge and the feedback is applied as-is to the cells. Since this balancing can only occur during resting windows, the current feedback is only applied during the resting phase and ignored on the other phases. When the resting phase ends, which the RL agent can't control the timings of, balancing no longer occurs.

For passive balancing each cell is balanced independently only during rest, at which point the current going through the cells at rest is defined by Equation (3.28). To include balancing, an additional balancing current $i_{balance}$ is added to the equation. The total current $i_{net}(t)$ flowing through cell i during passive balancing is as follows

$$i_{net,i}(t) = i_{balance,i}(t) + i_{self\text{-}discharge,i}(t) + i_{leakage,i} \tag{3.29}$$

Where $i_{net,i}(t)$ is the total current flowing through the cell, $i_{self\text{-}discharge,i}(t)$ is the self-discharge current flowing through the cell, $i_{leakage,i}$ is the leakage current of the cell and $i_{balance,i}$ is the balancing current of the cell. The balancing current is controlled externally to the simulation.

Active balancing is approached differently in this work. Because active balancing occurs during the discharge phase, cells almost constantly have current flow through them. With the balancing strategy proposed in this project, each different cell will have a different amount of current flowing through them, controlled by the power electronics. The power electronics will enable the balancing during operation strategy by allowing varying amounts of charge to be drawn from individual cells at any given time. For this, the balancing feedback is treated as a percentage that represents the percentage of power that each cell will provide based on the power demanded by the simulation profile. For example, if a cell pack is demanded a set amount of power from each cell, the balancing feedback can distribute the demanded power differently on each cell, allowing low SOC cells to provide less charge and high SOC cells to provide the rest of the charge. This way, the perceived power granted from the pack is the same, but each cell contributed in different amounts.

Active balancing with this work's approach only occurs during discharge. By modulating the power demand of each cell, the amount of charge each cell provides will be different, reflected by the discharge current i_{app_j} of each cell, as shown in Equation (3.13). In order to control the power of each cell, we introduce a control variable C, is obtained externally from the RL model, which will act as the balancing factor. Each cell will have a different control value C, and it directly affects the power usage P of each cell.

Normally, when there is no balancing, each cell will be demanded the same power P, and the pack power demanded will be $P \times N$, where N is the total number of cells. This must always hold true in order to provide the correct power demanded from the pack. As such, the control variable C must also respect this constraint, thus the following must always hold true:

For cell j, $j \in \{1, 2, \cdots, N\}$

$$\sum_{j=1}^{N}(P \times C_j) = P \times N \tag{3.30}$$

Where C_j is the control variable, obtained from the RL model, P is power demanded by a single cell, obtained from the drive cycle profile, and N is the total number of cells.

Additionally, it must not be possible for the control variable C of any cell to be 0. This case is not realistic and could cause potential harm on a real cell pack. For this, the control variable C must also always be in a safe range, which we defined as follows

For cell j, $j \in \{1, 2, \cdots, N\}$

$$0.5 <= C_j <= 1.5 \tag{3.31}$$

where C_j is the control variable of the cell.

With the control variable C defined, the discharge current i_{app} of any cell during active balancing is defined as follows

$$i_{app_j} = \frac{(v_{\text{OC},j} - v_{\text{RC},j}) - \sqrt{((v_{\text{OC},j} - v_{\text{RC},j}))^2 - 4(P \times C_j)R_{0,j}}}{2R_{0,j}} \tag{3.32}$$

where i_{app_j} is the discharging current, P is the power demand for a single cell, obtained from the drive cycle profile, C_j is the modulating control variable, obtained from the RL model, $v_{\text{OC},j}$ is the open circuit voltage, $v_{\text{RC},j}$ is the RC circuit voltage, and $R_{0,j}$ is the resistance in series.

The balancing feedback for each cell is not controlled or changed in the simulation itself, thus it must be provided correctly externally since it will be applied as-is.

3.3 Reinforcement Learning Model

This section describes the reinforcement learning algorithms and methods devised to train on the battery pack simulation.

3.3.1 RL environment

The simulation environment is interfaced with the RL agent via action space and observation space. The action space represents the parameters of the environment which can be directly altered by the RL agent.

We have two different action spaces for each balancing topology. For passive balancing, the action space is the balancing current, the additional current discharged for each cell by the BMS. The additional balancing discharge current values are between 0 and 1 Ampere, where 0 means no additional discharge current. For the reconfigurable active balancing during discharge, the action space represents the usage percentage for each

cell, an abstraction of the duty-cycling of the power electronics that link each cell. The values for the usage percentages are between 50% and 150%, for each cell. The total usage values of the cells must always provide the same total power to the EV, according to the drive-cycle demand. Vehicle power demand does not get altered by the topology and controller, they can only alter the way power is drawn from the cells by requesting more power from some cells, and less from others to compensate.

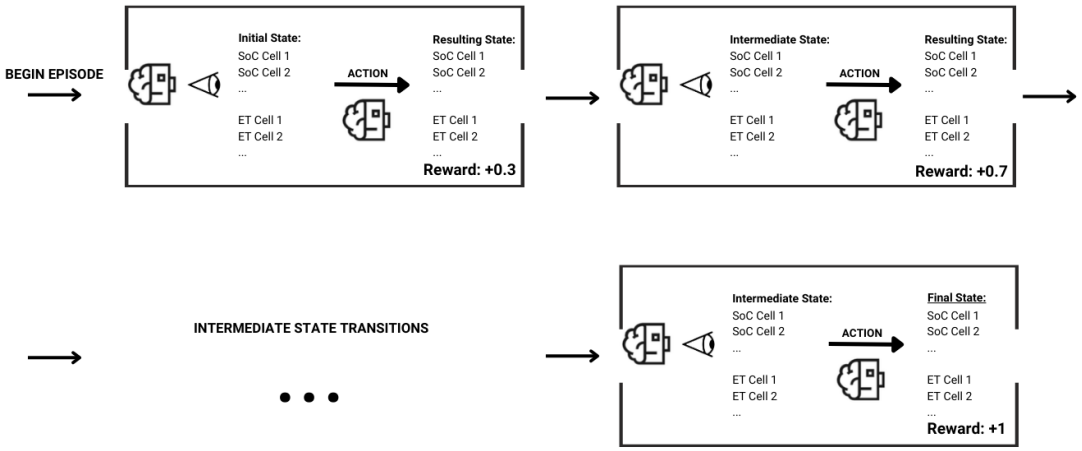


Figure 3.6: State transition within a training episode

Both topologies utilize the same observation space, which consists of the SOC values and the energy throughput of each cell in the pack. Each state represents an instance of the observation space, seen in Figure 3.6. The agent observes a given state, chooses an action according to its latest policy, and broadcasts the action which is then applied to the environment, leading to the following state. This state transition is then assigned a reward according to the reward function. The process continues until the transition to the final state.

State transitions in RL are designed to follow the Markov property, outlined in the Markov property below, Equation (3.33).

$$P(X_{n+1} = x_{n+1} | X_n = x_n, \ldots, X_1 = x_1) = P(X_{n+1} = x_{n+1} | X_n = x_n), \forall n \in N. \tag{3.33}$$

The Markov property asserts that the current state of the environment encapsulates all relevant information needed to determine the future state, rendering the history of previous states and actions unnecessary. Formally, in a Markov Decision Process (MDP), the probability distribution of transitioning to the next state from the current state, conditioned on both the current state and action, remains independent of past states and actions. This property simplifies the learning process by allowing RL agents to make decisions based solely on the current state, facilitating more efficient and scalable algorithms [29].

The reward function must take this property into consideration, and only assign rewards for the transition between one state to the next, according to an intermediate action. Individual state-transition rewards must not take into account past or future rewards. The elements for such behaviour are handled at higher levels within the RL algorithm, and not within the state-transitions themselves.

3.3.2 Action sampling rate

As the sample generation is bound by computational power in online RL, we make efforts to optimize the sample generation speed, at the cost of granularity in actions. Instead of computing a new action for each subsequent timestep of the simulation, we apply an action for multiple timesteps before interrupting and receiving the resulting state and assigning the reward for the transition, according to the reward function.

The frequency at which a new state is produced depends on the sampling rate of the simulation. All states are linked together by actions, except the initial state of an episode. An episode is a full run of a simulation, for a set amount of cycles.

For our balancing objective, we simulate many charge-discharge cycles across the lifetime of a pack within a vehicle. In order to reduce the computational load, by limiting the interrupts to the simulation for additional action commands from the RL agent, we sample the environment and return an action once every 30 in-simulation seconds, which constitutes a sampling rate of 30. Thus we reduce the time needed to complete a simulation by reducing the amount of context switching.

This sampling choice also affects how the reward function interacts with the environment and shapes our learning objective. As actions last for several timesteps, aggressive BMS actions such as large discharge currents can make the cells unbalanced, which the RL agent learns to account for. Indeed, as a tangential benefit, this acts as a limit to high discharge rates in the case of the passive topology. As high C-rates are undesirable in battery systems because they significantly lower lifetime of a pack. The RL agent is assigned 30 timesteps in which to gradually discharge the battery to a balanced state, rather than attempting to do so on a second-by-second basis, which experimentation has shown will consistently lead to the usage of maximum C-rates between seconds if not included as part of the reward function.

The C-rate limiting offered by this approach reduces the complexity of the reward function, which allows for much more reliable discovery of optimal policies. As the complexity of the reward function increases, the difficulty for the algorithm to discover relationships between state parameters and actions also increases.

When rewards are delayed, however, it can become challenging for the agent to understand which actions contributed to the eventual reward. This is known as the temporal credit assignment problem. Methods like eligibility traces in TD learning help mitigate this by attributing rewards to actions taken earlier.

3.3.3 Reward Design

There is no guarantee of convergence to an optimal policy for RL algorithms, and each new training cycle can lead to different results. An improperly defined reward function can easily lead to poor results. Achieving a truly optimal policy is better defined as an aspirational goal rather than a guaranteed outcome.

'Frequency' and 'scale' matter greatly for actions and rewards, as they directly influence the learning process and the efficiency of the policy. If some rewards are scarce, the agent will never discover how to get that reward and develop the optimal policy.

The online nature of reinforcement learning makes it possible to approximate optimal policies in ways that put more effort into learning to make good decisions for frequently encountered states, at the expense of less effort for infrequently encountered states [29]. Rewards that appear in states with low frequency are called sparse rewards. In environments that only provide sparse rewards, the agent struggles to learn effectively because it receives little information about the states at all times, and cannot infer what what constitutes a good or bad action.

An example of a traditional and simplistic RL reward function is a reward of $+1$ when all cells are balanced (i.e. when all cells are almost equal in SOC). The simulation starts, and the cells start off in a fresh, balanced state. Because of this, the agent gets rewarded in the beginning, and then stops getting rewarded the moment they unbalance. After unbalancing, the agent gets no feedback from the reward as to any improvements that are pushing the cells closer to a balanced state. None of it's actions can be quantified to an 'improvement'. It only receives $+1$ in the state of absolute balance, and 0 otherwise. Once the rewards become 0 from the unbalancing, the agent has no indicator that tells them if they are getting close to balancing again or not, and the exploration process fails. Whatever actions the agent takes after unbalance, they rely entirely on the random occurrence of cells reaching the 'balanced' state again, by chance. These events are so infrequent, that the cumulative changes in policy are insignificant. No policy can be learned from such a reward in this environment.

Policies are developed through improvement and iteration, the agent starts off with a bad policy and, through trial and error, gradually converges to the best policy. If we have no logical string of improving rewards to develop a policy, that offer a significant reward change relative to the rewards in the space, the agent cannot discover an optimum. If supplied only scarce 'maximum' rewards, the agent cannot infer the path to get to them.

Experimentation has shown that while using scarce rewards, during the first few steps of the simulation, any actions taken on the balanced state of the battery can predominantly lead to faster unbalancing. The only 'correct' action is inaction from the agent, and no further policy can be discerned as no rewards can point the agent towards restoring a balanced state.

As a reward cannot simply happen only during a balanced state, it must happen in stages and eventually lead the policy to the maximum reward. This technique is called 'reward shaping'.

Through reward shaping, we have experimented with reward functions which give the

agent 'hints' about how close it is to balancing the cells, through partial rewards. Maximum rewards are given once the objectives are fully achieved.

A common mistake made in reward shaping is to yet again disregard the frequency and scale of rewards. If we design a partial reward which is large, and does not lead to the maximum reward, the agent will remain focused on the partial reward and never evolve to the maximum reward.

In an experimental example, a reward which sums instances of $+0.1$ for each balanced cell can very easily get stuck on balancing only half the cells, or only 9 cells out of a total of 10. The reward significance from the states where all 10 cells are balanced is so small (0.9 vs. 1.0) that the policy fails to update. It does not acknowledge a variation of just 0.1 that happens with very low frequency to be significant enough to update the whole policy, because the discovered rewards are already strong and deviating from them will lose the perceived progress (effectively giving up exploitation), and thus exploration fails and the agent never learns to balance all 10 cells.

By introducing a second or third reward factor, such as energy throughput, new considerations must be taken. Challenges arise from the increased complexity in balancing multiple objectives and ensuring that the agent learns an optimal policy that appropriately considers all reward components.

Different reward factors might represent conflicting objectives. For example, cell balance and complete energy throughput uniformity are objectives that partially go against each other, depending on the characteristics of the cells and their behaviour during functioning. Balancing these conflicting objectives requires careful tuning to avoid preferential behavior where the agent consistently ignores one objective in favor of another. Introducing new reward factors can lead to unforeseen consequences. An agent might exploit loopholes in the reward structure to achieve high rewards without actually performing the intended task.

An example of such a loophole was found experimenting with purely negative reward structures, where the maximum possible reward per state is 0 and all other rewards are negative. Control theory has many overlapping concepts with RL, and is the preferred control method utilized in traditional BMS systems. In control theory, the equivalent for the 'reward' component of RL is cost. While RL typically aims to maximize cumulative rewards, control theory generally focuses on minimizing cumulative costs. A setpoint is established as the reference of the control system, for example in a PID (Proportional–integral–derivative) or MPC (Model Predictive Control) system. The cost gets higher as the measurements diverge from the reference setpoint, the desired values.

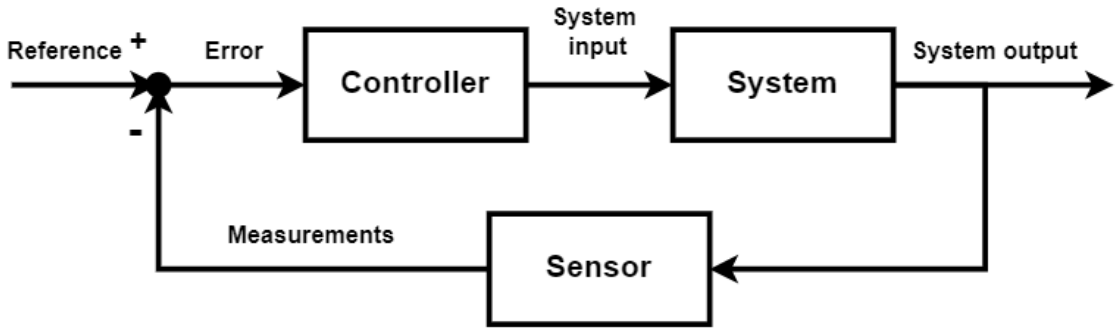


Figure 3.7: Control feedback loop

By testing the principles of cost in RL training and relying on purely negative rewards in experiments, several observations were made. The objective shifts to minimizing a cost, rather than maximizing a reward. As the agent gets 'punished' with more negative rewards the longer it remains in an undesirable state, the agent prioritizes quickly exiting the state. This introduces 'speed' as a primary factor of the reward function. However, as the agent quickly tries to escape the future 'punishments' given for each step spent in a disadvantageous position, it quickly discovers that it can forcibly end the simulation early and stop any future costs, by crashing the battery pack through over-discharge.

Thus the agent has 'minimized' the accumulated costs by stopping the episode from generating any additional cost-incurring states, by destroying the battery pack and finishing the simulation.

Cost-based behavior has been shown to be difficult to work around in an RL environment, and dangerous in a safety-critical vehicle application. The target environment of the agent must employ strict and thorough constraints in order to function on a cost-based behavior. Working with such a method requires hard constraints that the agent cannot violate, such as safety limits on state and action variables. These restraints would, as a result, heavily limit the possible actions of the BMS controller and the ability of the agent to achieve it's task using the full range of possible actions.

Applying a larger penalty at the end of a forced pack over-discharge has also not proven to be an effective method of avoiding such behavior. As stated previously, sparse rewards are not an effective tool in policy optimization. As the over-discharge state is the last state in the training cycle, the large lump reward will skew the policy into attributing the penalty as the result of that immediate action. Because the over-discharge crashing is discovered with very high frequency, as part of the learning process, the agent will consistently default to this behavior in a cost-based reward function, no matter the adjustments in cost weights or additional costs for crashes.

3.3.4 Action/Observation space normalization

For RL algorithms, effective exploration and learning depend on the appropriate scaling and normalizing of the action and observation spaces, as well as the issued rewards. Such normalization techniques ensure stable and efficient learning processes for RL agents.

The action and observation spaces in RL environments can vary widely in scale and magnitude, posing challenges for RL algorithms in learning policies and behaviors effectively. Normalization techniques aim to address these challenges by ensuring consistency and stability to these spaces. Proper normalization facilitates smoother convergence, improves exploration-exploitation trade-offs, and enhances the generalization capabilities of RL agents across diverse environments.

The functions which serve as the basic building blocks for RL algorithms are inherently sensitive to parameter scales. By scaling the state and action spaces appropriately, normalization techniques promote more effective exploration and allow the agent to discover promising regions of the state space more efficiently.

The Bellman formulas (Equations (2.5) and (2.6)) express the value of a single state in terms of the expected sum of rewards obtained by following a policy from that state. Proper normalization ensures that the values of states remain within a reasonable range during learning. This stabilizes the learning process, encouraging faster convergence. Normalization helps the agent generalize across different states and environments by ensuring that the learned values are comparable across varying scales and magnitudes.

Q-functions represent the expected cumulative rewards obtained by taking a specific action in a given state and following a policy thereafter. Normalizing action spaces ensures that actions are represented in a consistent and comparable manner across different environments. This helps in learning more stable Q-values and improves the agent's ability to evaluate the quality of actions accurately. Proper normalization prevents vanishing or exploding gradients during Q-learning updates, leading to more stable and efficient learning. Normalization also aids in generalizing Q-values across diverse state-action pairs, allowing the agent to transfer its knowledge to new situations effectively.

Normalizing input features ensures that the model encounters consistent scaled data during training, and convergence is impossible for most algorithms without it.

Within our RL framework for SOC values, which are originally between 0 and 100, we employ min-max normalization. In this process, the minimum value in the original range of 0 is mapped to 0, and the maximum value of 100 is mapped to 1.0, with all other values linearly scaled in between. This normalization technique ensures that the values fit proportionally within the desired range while preserving their relative relationships.

The energy throughput values go through a more complex process. As energy throughput values increase constantly throughout a simulation, being proportional with the current that courses through cells, we cannot rely on min-max normalization. We would require the maximum value of the energy throughput of a cell, in the final state. However, during the training process, we do not have access to this value until the very end of the simulation, and energy throughput must be already normalized and assigned to the intermediate transition-step states in order for the learning process to take place.

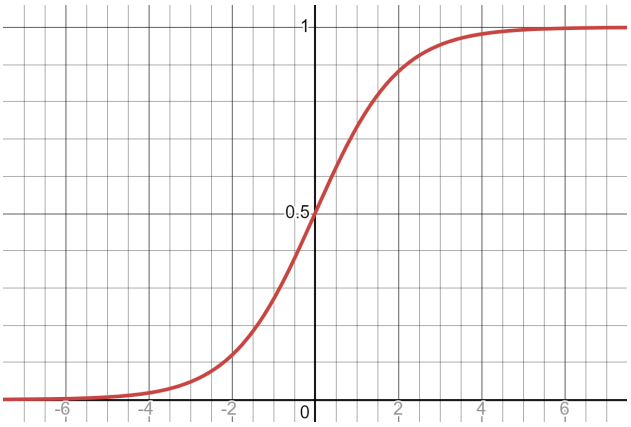


Figure 3.8: Logistic sigmoid curve

To normalize the constantly increasing energy throughput values, we opted for a modified sigmoid function that maps its input to the range [0, 1]. This modification is known as a "logistic function" or "logistic sigmoid function" shown in Equation (3.35). This function maintains the sigmoidal curve but changes the bounds of the values, seen above in Figure 3.8.

Before filtering the values through the function, we subtract the smallest energy throughput value in a given state from every single energy throughput value in that state, following the Equation(3.34). Thus, the smallest energy throughput cell will have the value 0, and the rest of the values represent how much more energy throughput the other cells have experienced in relation to one another. The goal, outlined by the reward function, is to reduce these differences as much as possible.

$$ETdiff(s,i) = \frac{ET(s,i) - minET(s)}{1000} \tag{3.34}$$

$$\sigma(ETdiff(s,i)) = \frac{1}{1 + e^{(-ETdiff(s,i))}} \tag{3.35}$$

The action space of an RL framework must also be normalized for proper functioning. When actions are on different scales, the gradients used for updating the policy can vary widely in magnitude, potentially causing unstable learning. Normalizing the action space helps in maintaining consistent gradient magnitudes, leading to more stable learning curves. Large or inconsistent action values can lead to numerical instability, especially in deep learning algorithms which are the target of our study. Gradient-based methods like policy gradient or actor-critic algorithms typically expect action values to be normalized to prevent large updates that destabilize learning.

Without normalization, actions with larger scales might dominate the exploration process, causing the agent to miss out on potentially valuable actions with smaller scales.

Normalizing the action space also allows for the transfer of learned policies across similar environments. This is because the normalized actions are less dependent on the specific scale in any given environment, making the policies more generalizable. This is a highly

desirable trait in the industry, as it allows a model trained for one battery pack to be used on a whole series of packs of the same type.

The BMS passive balancing currents are represented, initially, by values from $[0, 1.0]$. These are the additional currents drawn from the cells for balancing purposes. The power usage balancing topology instead represents the percentage power draw from a cell, between 50% and 150%, originally received by the simulation as $[0.5, 1.5]$.

Both balancing topologies action spaces are min-max normalized to values between $[-1.0, 1.0]$, and converted in to simulation by the transformations Equation (3.36) and Equation (3.37). Thus the RL actions maintain the same $[-1.0, 1.0]$ scale within the RL training section.

$$n_{passive}(action) = \frac{action + 1}{2.0} \tag{3.36}$$

$$n_{active}(action) = \frac{action + 1}{2.0} + 0.5 \tag{3.37}$$

Lastly, the rewards themselves must maintain the same scale for the same overarching reasons of consistency and preventing extreme updates. The rewards that can be gained within a state-transition must designed in such a way that they are bound within a value range. Using normalized and bounded rewards, the agent is less likely to get stuck in local optima driven by disproportionately high or low rewards, thereby promoting better exploration of the state space.

Should the values remain unbounded, the rewards skew their significance. For example, if a reward function is designed to output maximum values of 10, but has unbound lower values, it creates a large imbalance in the reward distribution. The negative rewards with extreme values disproportionately influence the learning process and are assigned greater significance in the policy updates.

Taking all of the previous observations and methods into consideration, we arrived to the following reward formula. The objectives of the RL agent are to balance the SOC values of cells and attempt to maintain uniform energy throughput simultaneously.

The reward is split into two segments, which alternate based on a threshold. In the first segment, the SOC balancing is rewarded by itself, without any energy throughput considerations, shown below in Equation (3.38).

$$R_{SOC}(s, a) = 1 - (\frac{\sum_{i=0}^{nr\_cells-1}(SOC(s, i) - minSOC(s))}{nr\_cells}) \times k \tag{3.38}$$

The maximum reward achievable for this segment, in a single state transition, is 1. From this value, we subtract the sum of the differences between the SOC values of the cells and the smallest SOC value in the pack. The smaller the differences in SOCs between cells, the more the reward will be increased. The sum term is then divided by the number of cells, having a maximum value of 1, reducing the value of the reward to a minimum of 0 once differences are too high. Lastly, the variable k can take values between 1 and

10 to increase the significance of lower SOC differences, and lower the minimum reward. The k value 10 was shown to offer the best sensitivity to SOC differences in our models.

The next stage of the reward, shown below Equation (3.39), is unlocked once the largest SOC difference has gone below a set threshold. Some usual thresholds used in experiments are 3.5% and 5% (i. e. all SOC differences to the smallest SOC are lower than 5%). Through this approach, the RL agent is allowed room to maneuver and attempt to achieve uniform energy throughputs. The agent maximizes SOC rewards once all SOC differences are below the threshold.

$$R_{ET}(s,a) = 2 - (\frac{\sum_{i=0}^{nr\_cells-1} ETdiff(s,i)}{nr\_cells}) \tag{3.39}$$

Once the SOC threshold has been satisfied, the second stage of the reward includes the full maximum reward of the previous stage, which covers SOC balancing. Additionally to this reward we add 1 and subtract the sum of normalized energy throughput differences. Lastly, we divide the term by the number of cells, reaching again a maximum value of -1. Thus, in a state where SOCs have been balanced to the threshold, but the energy throughput values are completely non-uniform, the energy throughput reward will reach the minimum of 0, and the total reward, adding the SOC segment, will be 1. The final reward function is formalized below in Equation (3.40)

$$R_{SOC+ET}(s,a) = \begin{cases} R_{ET}(s,a), & SOC\_diff < threshold \\ R_{SOC}(s,a), & SOC\_diff \geq threshold \end{cases} \tag{3.40}$$

where $threshold$ is the percentage threshold of SOC_diff previously described.

This method of gradually progressing to more complex tasks is known as curriculum learning. We take this segmented approach in order to make the SOC balancing element of the objective easier to discover. Instead of the RL agent immediately getting a complex reward that varies with both energy throughput and SOC balance, the rewards are introduced gradually. It allows the agent to master the simpler objective before beginning to learn the complex combination of objectives.

Most importantly, however, the threshold-based rewards allow the agent the freedom to pursue energy throughput uniformity with a higher degree of freedom once the desired SOC limit has been reached. The complex combined reward is introduced only after SOC balancing has discovered a suitable policy, upon which the SOC-ET aware policy can then be built and further iterated on towards the final optimal policy. This approach has shown to significantly improve reliability in reaching convergence, and the stability of the training process.

3.3.5 Algorithm analysis

Model-based RL approaches require certainty that the approximation fully captures the essence of the environment, one that is identical to the ground-truth. It is also crucial for

this approximation to be very efficient in generating predictions, otherwise the time-to-train becomes infeasible. These approaches work best with easily modeled environments with few or no non-linear factors, such as games of Chess and Go [32].

Model-free approaches are applied directly to an environment, and learn through trial and error. A model-free agent must take an action, and see the results, to be able to learn from it. This grants much more flexibility and eliminates the risks of inaccurate models training unusable agents. It comes with the cost of less sample efficiency, as we need to run more simulations for the agent to find the best course of action.

We cannot directly measure the charge of a battery pack during usage. The measurements of a battery rely on estimations. Battery pack charge is difficult to estimate with accuracy even within an isolated lab environment, and much less so in within a functioning electric vehicle fitted with down-scaled sensors. There are many unpredictable factors that require modeling to ensure fidelity to ground-truth, ranging from the position of a cell in the pack, composition of cooling mechanisms, weather conditions, etc.

Trying to create another layer of abstraction by designing a predictive model for the RL agent would not be only infeasible, but also counter-productive. The more we move away from the ground-truth of the battery pack, the more overfitted our solution becomes to a certain pack topology, cell chemistry, and estimation method, and the risk of developing agents which fail when tested on real-life applications grows exponentially.

Model-free methods are the most suitable for our cell control task, as they offer a high degree of decoupling from the battery pack environment. The simulation can be easily modified for entirely different battery pack setups, without breaking the training framework.

Within the model-free we examined both Q-learning methods and Policy Optimization methods.

The first algorithm of the Policy Optimization class was the Policy Gradient, which then evolved into Deep Policy Gradient (DPG) when deep-learning methods were introduced. The current state-of-the-art policy-based algorithm is Proximal Policy Optimization (PPO), known for its stability and ease of use, able to learn most environments without any hyperparameter tuning. It is currently the preferred algorithm of OpenAI and many other research groups within the field.

Within our battery pack simulation, the action space is represented by the duty-cycle regulated power usage percentages of each cell. They are, by definition, continuous variables and thus pure Q-learning approaches, such as DQN, cannot be effectively applied without gross approximations which eliminate the advantages of a precise power-electronics topology. More importantly, pure, non-DL Q-learning is a tabular method which does not scale in high-dimensional environments. A battery pack may have up to several hundred cells, each with its own action-space variable and observation-space parameters.

DQN methods cannot train agents in continuous environments without approximations, as attempting to compute argmax over a continuous function is not feasible. Among such approximations there are, for example, fuzzy-logic implementations. The contin-

uous action space is partitioned into discrete fuzzy sets, each representing a range of continuous actions. These sets use membership functions to determine the degree to which a continuous action belongs to a fuzzy set. This allows the agent to select actions based on fuzzy values instead of fixed discrete actions. However, this discretization process introduces approximation errors, especially when the action space is large or highly complex such as a battery pack. The granularity of the fuzzy sets does not accurately represent the underlying continuous actions, leading to suboptimal decisions by the agent.

However, the benefits of sample efficiency in Q-learning are thankfully not forfeited. Q-learning and Policy Optimization are not mutually exclusive, and there exists a third class of hybrid algorithms which make use of a combination of these design philosophies.

The first iteration of these algorithms is DDPG (Deep Deterministic Policy Gradient). It combines ideas from DPG (Deterministic Policy Gradient) and DQN (Deep Q-Network) to create a powerful approach capable of handling the complexities associated with continuous control tasks. It utilizes an actor-critic framework, where the actor-network directly learns a deterministic policy mapping states to actions, and the critic network estimates the action-value function (similar to Q-learning).

DDPG is a foundational algorithm in the domain of continuous action reinforcement learning, and similarly to our previous algorithms, it has evolved into two key derivatives: Twin Delayed DDPG (TD3) and Soft Actor-Critic (SAC).

TD3 uses two Critic networks (and their target networks) and takes the minimum value of the two Q-values to calculate the loss for the Critic. This approach reduces the overestimation bias present in DDPG. TD3 updates the policy (and target networks) less frequently than the Q-function updates, which helps in reducing per-step variance. As TD3 trains a deterministic policy, noise is added to the target action to ensure exploration during training.

SAC, another derivative of DDPG, tries to improve sample efficiency by incorporating entropy into the reward objective, encouraging exploration naturally through the policy. Like TD3, SAC uses two Critic networks to mitigate positive bias in the policy improvement step but integrates this within an entropy-maximizing framework. The entropy factor has to be tuned according to the environment, otherwise it can disrupt the learning process.

Both variants share a common ancestor and implement similar mechanisms, with the main differences arising in their exploration method.

For the purposes of a battery pack controller, we have found TD3, SAC as well as PPO to be suitable to be considered for the task definition, and we will outline the trade-offs of these algorithms. We will be using the stable_baselines3 implementation of the Open AI specifications for these algorithms.

3.3.6 PPO vs. TD3 vs. SAC

First, the nature of the policies within these algorithms must be examined. PPO and SAC use stochastic policies. A stochastic policy is one where the action chosen by the policy

is not deterministic but probabilistic. Instead of always choosing the same action in a given state, a stochastic policy outputs a probability distribution over possible actions. The action is then sampled from this distribution, meaning that even in the same state, different actions may be chosen at different times. Randomness is thus introduced in the action selection, and used as a vehicle for exploration.

Stochastic policy gradient algorithms generally work by sampling from a stochastic policy and adjusting the policy parameters to increase the cumulative reward. The policy depends on parameters are the weights and biases of a neural network, and the algorithm attempts to find the optimal parameters that maximize performance gauged by rewards.

Stochastic policies are valuable during the training phase of RL. They enable the agent to explore a wide range of actions and avoid local optima by sampling actions from a distribution rather than choosing a fixed action. An example of probability distribution, the normal distribution, can be seen below in Figure 3.9.

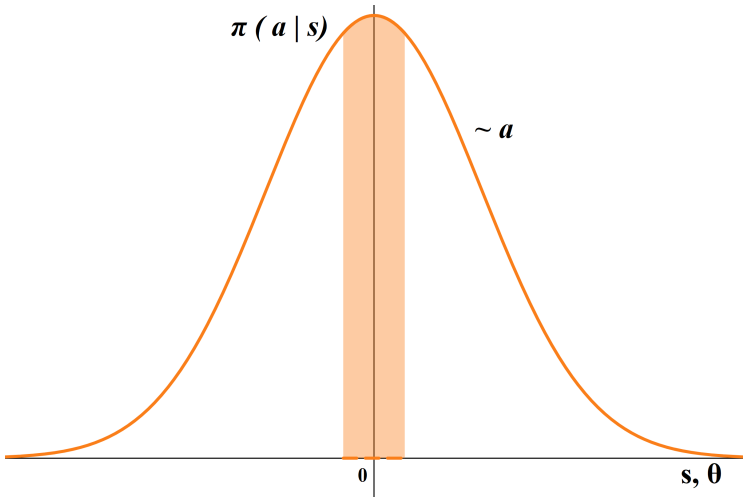


Figure 3.9: Probability distribution - normal (Gaussian) distribution

TD3 is deterministic by nature, and it is a defining feature of the algorithm. This means that, given the same state input within the same environment, a TD3 agent will output the exact same actions and replicate results consistently. In order to achieve exploration, TD3 introduces action noise, typically Gaussian, to try out new paths during training.

The agent's actions in a deterministic policy are relatively predictable and consistent. Given the same state and policy parameters, the agent will always produce the same action. This consistency is crucial for tasks where reliability and predictability are important, such as battery pack control systems.

During deployment, the noise added for exploration is completely removed. In the final deployment the policy outputs the action directly from the deterministic neural network without any added noise.

The pure deterministic nature of TD3 can lead to smoother action trajectories compared to stochastic policies used in PPO and SAC, where actions have some level of randomness. Additionally, in TD3, two separate Q-functions (Q-networks) are used to estimate the

action values, using a delayed policy update mechanism. The policy is updated less frequently and this helps stabilize the training process and can lead to smoother policy updates over time.

An agent that exhibits stochastic behavior, where the same inputs can lead to different outputs, poses significant challenges for testing, validation, and certification. This unpredictability makes it difficult to ensure consistent performance and to meet stringent safety standards. However, there are methods by which stochastic policies, once fully trained, can be adapted to mimic deterministic behavior.

Once a stochastic policy has been trained to convergence, the exploration phase can be considered complete. At this point, the policy should have identified the most optimal actions for given states, reducing the range of probabilities. By reducing the randomness, the stochastic policy can have outputs similar to deterministic outputs, albeit it in an approximate way.

A more thorough and final method of such a conversion from stochastic to deterministic is to only sample the highest probability action and discard the rest of the distribution. Thus the stochastic policy is definitively transformed into a deterministic one. Another common approach is to use the mean or mode of the action distribution as the final action. For instance, in a Gaussian policy, the mean action (the peak of the distribution) can be used, effectively making the policy deterministic.

Next, we consider how learning takes place. All algorithms train a policy, however, the two RL design philosophies for developing said policies are 'on-policy' and 'off-policy' respectively.

PPO utilizes on-policy training, which optimizes the policy by directly maximizing a clipped surrogate objective function, which balances between policy improvement and similarity to the old policy. This approach ensures that policy updates are constrained to prevent large deviations. By utilizing only data collected from the current policy, PPO ensures consistency between the policy being learned and the experiences used for learning.

As a trade-off for this consistency, PPO is only able to use data acquired through interactions with the current latest, best, policy in the training process. Interaction data from previous versions of the policy is discarded and not used during the next iterations of training. This causes PPO to be much more sample-inefficient than the off-policy alternatives SAC and TD3.

TD3 and SAC are on-policy algorithms. This type of algorithms are defined by their sample efficiency. The efficiency stems from their ability to reuse past experiences from previous iterations of policies during training, by storing the interaction data in a replay buffer. This leverages the usage of historical data during training and significantly reducing the number of required additional interactions with the environment.

The policy of off-policy algorithms effectively learns from its previous versions and becomes much less reliant on fresh datapoints. By contrast, PPO is entirely reliant on continuously pulling new environment interactions with its latest policy.

The sample inefficiency of PPO has lead to more than double the number of required

state-action transitions in order to reach similar progress. A trained PPO algorithm averaged 6 million training steps required for convergence. PPO stands as one of the most widely adopted reinforcement learning algorithms, primarily due to its ease of implementation, insensitivity to hyperparameters, and stability in training [18]. This has made the algorithm a popular choice for RL applications in the field of battery pack control. However, experiments have indicated significant limitations when applying PPO to environments characterized by high-dimensional continuous action and observation spaces. When applying PPO for packs with more cells (i.e. 10 cells) for the action space, the algorithm struggles to converge to an optimal policy, due to the number of complex non-linear relationships between the parameters of the cells. The insensitivity to hyperparameters means that there is little hope of achieving convergence through hyperparameter tuning.

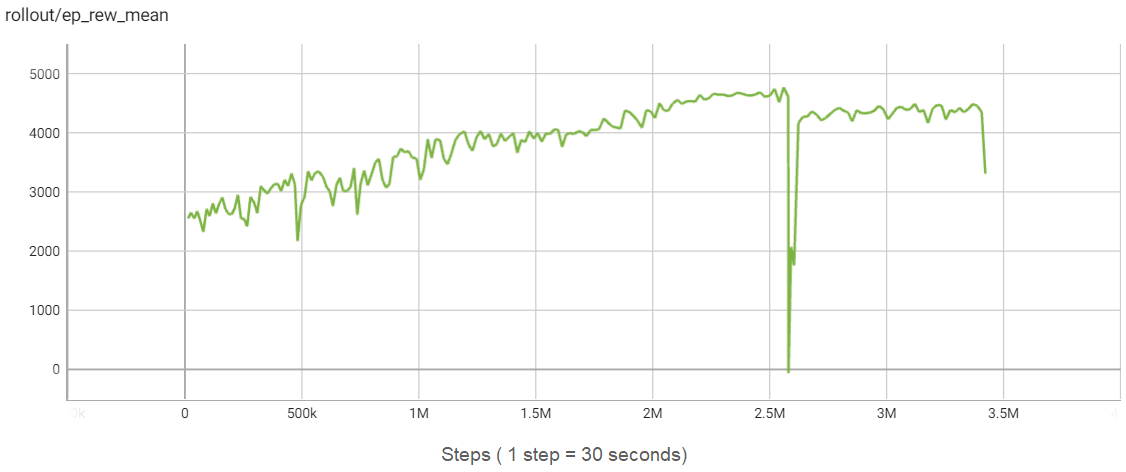


Figure 3.10: SAC mean episode reward - 2.5 million training steps to convergence

TD3 and SAC do not suffer from the dimensionality problems of PPO, and have been shown to handle large environments with many cells. These algorithms share the same common ancestor algorithm, DDPG, and share the same sample efficiency between each other. They differ, however, when it comes to stability in training. TD3 has shown a much less stable training process when compared to SAC.

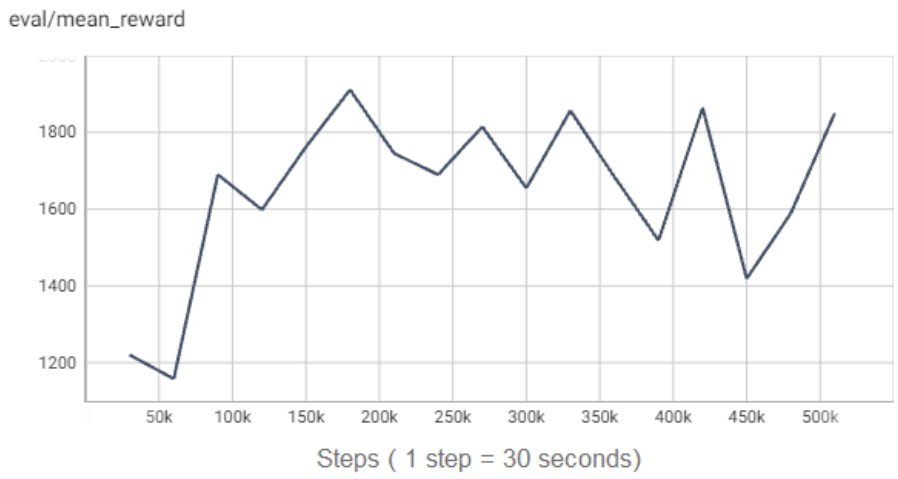


Figure 3.11: TD3 training instability - evaluation episodes

TD3 introduces several mechanisms to stabilize the Q-function updates, such as using two Q-networks to mitigate overestimation bias and delaying policy updates. However, these mechanisms are still prone to instability if the Q-value estimates diverge or fail to accurately represent the expected returns, as seen in Figure 3.11. SAC benefits from its use of a stochastic policy and entropy regularization, which combined lead to smoother updates of the Q-function. The added entropy term in the SAC objective function acts as a regularizer, discouraging sharp changes in the Q-values and promoting more stable training than TD3.

TD3 primarily focuses on minimizing the deterministic Bellman error, which, while traditionally effective, does not incorporate an additional mechanism to control the smoothness of policy updates. SAC simultaneously optimizes for both the expected return and the entropy of the policy, providing a dual objective that naturally balances exploration and exploitation. This balance helps maintain stability during training by preventing premature convergence to deterministic policies.

Through it's advantage in training stability, coupled with the ability to handle high dimensional spaces in a sample-efficient way, SAC was proven to be the best fit for our requirements out of the three algorithms. The final trained RL agents used throughout the paper use the SAC algorithm.

3.3.7 Training techniques

The battery pack simulation parameters used during the training processes are listed below, in Table 3.3.

Table 3.3: Simulation parameters

Parameter	Value	Description
cellModel	P14	Datasheet used for the cells used within the pack
profile	us06	Drive-cycle profile used for battery current demand
utilization	[10, 2]	Percentage of resting period in a episode, and duration of resting periods in hours
num_cells	10	Amount of cells simulated within a pack
simCycles	25	Number of charge-discharge cycles within an episode of the simulation
sampleFactor	30	Number of simulation timesteps until a state is returned and a new action is requested
seed	1	Seeded randomness of variable cell characteristics

All RL agents were trained on the P14 cell type using the 'us06' drive-cycle profile.

Several techniques were employed in order to improve training stability, one of which is curriculum learning. Curriculum learning involves training an agent by presenting tasks or environments in a sequence that progresses from simple to increasingly complex challenges.

By initially focusing on simpler tasks, the agent can learn core concepts and behaviors more quickly, which can then be transferred to more complex tasks. This leads to more efficient learning and often faster convergence. This principle can also be found in the reward function we use for out algorithms, and was expanded to the target environment itself.

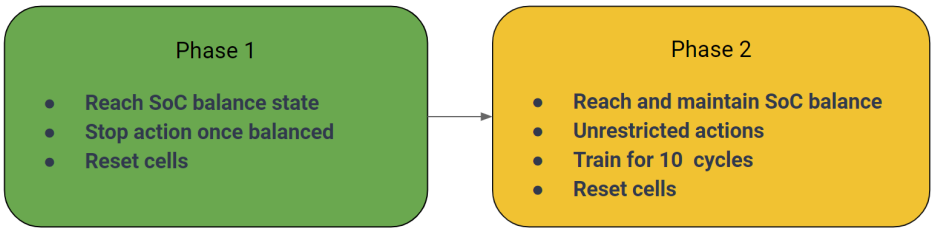


Figure 3.12: Curriculum learning environment phases

We first try to teach the agent the basics of SOC balancing. The goal is to have the cells reach a balanced state, once. We begin with an environment that prevents the agent from issuing further actions once it has reached a balanced state for the cells.

Next, we train this same agent to maintain balance throughout 10 cycles. We remove the action limitations. The agent is now free to take any actions on the environments, at any time. At the end of each training episode after all cycles are finished, for both environment phases, the battery pack is reset to initial charge values.

A third environment phase was then proposed, where battery charge reset functionality is removed and the training takes place in an uncapped amount of charge-discharge cycles.

However, testing has shown that this type of environment is infeasible for training. As the policy trains, even after having developed adequate balancing techniques in the previous phases, random exploratory actions can still push the pack into a state of over-discharge. Once the simulated pack reaches a state where cells are heavily imbalanced from a bad series of exploratory steps, the simulation cannot recover without a full cell charge reset.

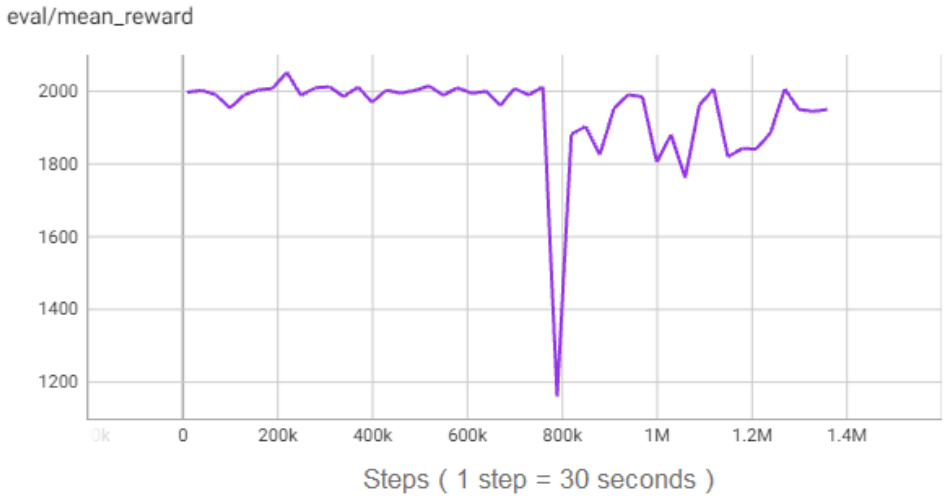


Figure 3.13: TD3 - Diverging after convergence

During training, especially in the case of TD3, policies may diverge while iterating towards the optimal policy, and the total reward returns per episode may spike up and and down drastically, instead of having a steady increase. Additionally, after converging the optimal policy, further training has a chance to lead to divergence and the collapsing the policy, pictured above in Figure 3.13 and Figure 3.10.

To mend these failures we use a callback function that activates after a certain amount of training steps within a training run. The callback function runs several evaluation episodes during which the policy and is tested, without training and altering the policy during those steps. The average total reward value of the evaluation episodes is compared with the previous best average, and if it is larger, the new policy is saved in a checkpoint as the 'best policy'. The training then continues as normal, and should further training fail to yield better results or lead to divergence, the best version of the policy found during that training run remains saved as a backup. Training can then be continued starting from the backup.

The final parameters used for the callback function can be found below in Table 3.4, alongside the ranges of values used during experimentation. The number of optimal evaluation episodes and evaluation timestep frequency depends heavily on the simulation parameters.

Table 3.4: EvalCallback function parameters

Parameter	Value	Range	Description
n_eval_episodes	2	[1, 10]	How many evaluation episodes should be run before computing the average total reward
eval_freq	30000	[1000, 100000]	Evaluate the agent every 'eval_freq' state-transition steps
deterministic	False	NaN	Whether to use stochastic or deterministic actions during evaluation
render	False	NaN	Whether to generate a visual render of the battery environment during evaluation
verbose	True	NaN	Print information about evaluation results on callback

The number of cycles, balancing topology and resting sequences determine the definition of an episode. Passive balancing outputs fewer timesteps, as it only acts during resting phases, while the reconfigurable power-usage balancing topology acts throughout the discharge phases and broadcasts many more states and requests for actions from the RL agent during training. More timesteps, combined with the higher computational demands of some algorithms leads to longer episode times. This requires a balance to be struck between the frequency of evaluations and the number of evaluation episodes that can be afforded during a training cycle, which must be tailored to the simulation and algorithm used.

3.4 Training Data

A training log of the SAC SOC+ET balancing algorithm can be found below:

Table 3.5: SAC training data

Variable	Value
ep_len_mean	4.2e+03
ep_rew_mean	4.71e+03
episodes	924
time_elapsed	66510 seconds
converted_time	18hrs 28min 30sec
total_timesteps	3268050
actor_loss	-113
critic_loss	0.182
ent_coef	0.00297
ent_coef_loss	-8.97
n_updates	19123374

Training times for sufficient convergence reach upwards of 18 hours. The -8.97 negative value of the entropy coefficient loss suggests that the actions under the policy have a relatively low probability, meaning the policy is still exploratory. The ent_coef value dynamically according to

The critic loss of 0.182 indicates that the mean squared error between the predicted Q-values and the target Q-values is relatively small, indicating the policy is making steps towards convergence. The large negative value for the actor loss of -113 indicates that the policy is choosing actions that yield high Q-values (expected returns) while also considering the entropy bonus. Essentially, the policy is performing averagely under the SAC objective.

3.4.1 Hyperparameters

Hyperparameters are the adjustable variables in ML/RL algorithms that govern the learning process and model behavior. Unlike model parameters, which are learned during training (e.g., weights in a neural network), hyperparameters are set prior to training and control aspects such as learning rate, batch size, and the number of layers in a neural network. Proper tuning of hyperparameters is crucial for optimizing model performance and achieving the best results on a given task. Properly selected hyperparameters can vastly improve training performance and improve convergence rate for less-stable algorithms.

Hyperparameter optimization can be automated via the use of algorithms to systematically search for the best hyperparameter values that maximize model performance. This process aims to find the optimal set of hyperparameters without manual intervention. However, due to the high training times, automated, random or grid, optimization was not feasible.

As a result, the components of the algorithms were studied individually and a manual tuning process was undertaken. Below we will list the most impactful hyperparameters that were present in all tested algorithms.

Learning rate affects how substantial the updates to policies are. A high value may improve training speed, but vastly increases the chances of 'overshooting' the optimal policy. The agent makes large updates to the policy when it sees a small improvement. If the environment is simple and straightforward, then we can usually apply a high learning rate. A lower value can vastly improve stability, while also increasing training times. With a lower learning rate, we ensure that we do not overshoot the optimal policy and decrease the occurrence of divergence.

Deep neural networks serve as function approximators, enabling agents to operate effectively in environments with high-dimensional and continuous state and action spaces. These networks predict the value of actions from given states or directly determine the best action to take, embodying policy-based approaches. They manage to capture the complex relationships and patterns within the environmental data, learning optimal strategies over time through gradient-based optimization methods.

A deep-learning network must be tailored to the objective of the algorithm. An improper network cannot discover sigificant relationships between features properly, and fails to converge. A network that is too small has no space to track meaningful clusters of features and is unable to learn patterns between them, while a network that is too large can overfit to insignificant features which do not apply to every battery pack, and render the trained agent unusable.

The exhaustive list of hyperparameters, networks, and filters used for the RL algorithms are listed below in Table 3.6.

Table 3.6: Hyperparameters used within the experiments - SAC

Parameter	**Value**	**Description**
learning_rate	0.0003	Learning rate of ADAM optimizer
buffer_size	2000000	Size of the replay buffer
batch_size	256	Percentage of resting period in a episode, and duration of resting periods in hours
tau	0.0005	Polyak soft-update coeficient
gamma	0.99	Discount factor
gradient_steps	1	Number of gradient steps after each rollout
action_noise	none	Type of action noise used, 'none' for entropy-based exploration
ent_coef	adjusted dynamically via optimizer	Entropy regularization coefficient, controlling exploration/expoitation
ent_coef_optimizer	ADAM	Dynamically adjusts entropy coefficient during training, ensuring policy remains sufficiently stochastic
target_entropy	-1	Target entropy when learning, determined by action space dimension
net_arch	[256, 256, 256]	Dimension of policy and value networks
action_space	Box(-1, 1, (10,), float32)	Environment values directly altered by the RL agent
observation_space	Box(0, 1, (2, 10), float32)	Environment values indirectly influenced by the RL agent
activation_fn	ReLU	Activation function
clip_mean	2.0	Clipping of mean output when using gSDE
features_extractor	FlattenExtractor	Features extractor used
n_critics	2	Number of critic networks

3.5 RL Tooling

The RL training framework is developed on Jupyter Notebooks on IPython Kernel using Python, version 3.11.7 for the algorithm training framework.

The RL algorithm implementations are provided by the stable_baselines3 project, an RL algorithm library based on the OpenAI Baselines. They provide a set of high-quality implementations of the state-of-the-art RL algorithms currently being researched and used in industry [35]. The training environments used to interface with the simulation are built using Gymnasium API, a community-maintained fork of OpenAI's Gym. The action and observation spaces are modeled using Gymnasium Box spaces. A Box represents the Cartesian product of n closed intervals [36].

The Tensorboard suite is used for graphing and visualization of algorithm training data logs, for variables such as loss values and mean reward which are used in evaluating the training process [37].

3.6 MATLAB-Python Interfacing

For this project, the cell simulation environment runs on MATLAB while the RL model training runs on Python. The cell simulation code, leveraged from previous work, was used as-is and an interface needed to be created to allow for the simulation of the cells to be controlled by the Python RL model, for training and testing purposes.

For communication between processes, UDP, TCP, and Memory-mapping (Memmap) interfaces were tested. For the final implementation, Memory-Mapping was utilized given that UDP and TCP were too slow for the amount of data that needed to be transferred between the processes. Despite the UDP and TCP interfaces not being utilized, they serve as a base for future implementation through the web that can allow for the cell simulation and RL training environment to be executed in a distributed manner.

Memory-mapping is a mechanism that maps a file onto RAM, allowing direct access to it without having to access the disk. By completely avoiding accesses to disks, as well as allowing for multiple processes to access the same mapped file, it can expedite and simplify the task of transferring data [38]. Memory-mapping was chosen as opposed to shared memory in order to reduce development time as well as the operating system where simulation and training were done is Windows, which has less flexibility for shared memory between processes compared to Linux.

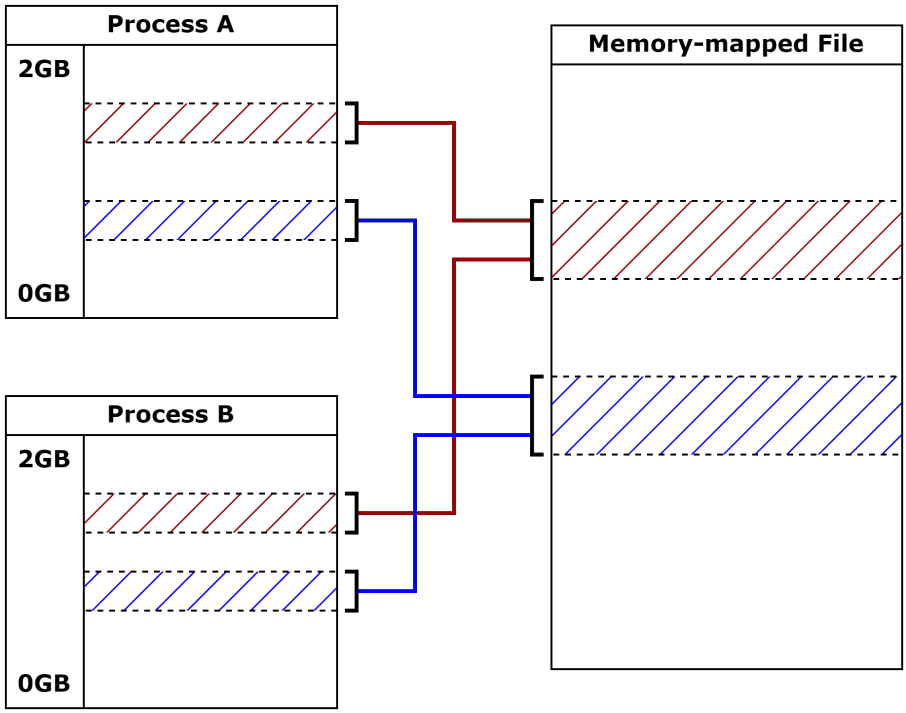


Figure 3.14: Illustration of two different processes sharing memory through a memory-mapped file.

3.6.1 Memory-map Interface Implementation

Through testing and by utilizing the memory-mapping packages of MATLAB and Python, it was possible to create Python libraries `mmapTx.py` and `SimCellPack.py` that handles the creation of the files that are used for memory-mapping, as well as initialization, handling and controlling of the cell pack simulation. The Python package used is `mmap`, executed on Python 3.9.18. The Matlab library used is `memmap` for MATLAB R2021b Update 6.

The library `mmapTx.py` was developed to create the interface using the Python memory-map library, and handles file creation as well as provide writing functionality and a blocking read functionality. It was designed to create two files, which are expected to be used by two different processes in an interlocking manner. A process can write to its output file as much as it wants, and when it wants to make that data available to the other process, it must set a flag. On the other hand, if a process wishes to read data from the input buffer, it must wait for the other process to set a flag that will allow it to read the data.

For the Matlab simulation code, the code directly uses the MATLAB-provided memory-map functionality to pass data through the memory-mapped files in a bare-bones fashion, with no library for the data transferring functionality. During the simulation initialization code, the memory-map interfacing is initialized as well and used directly. When the code reaches a point where is needs data from the Python process, it will be stuck on a busy-waiting for-loop until the Python process updates the necessary flags.

In order to pass data between the Python process and the MATLAB process, two files are created that will be utilized as memory-mapped files: `memmap_py_to_mat.txt` and `memap_mat_to_py.txt`. Each file will be considered half-duplex for data sharing, where file `memmap_py_to_mat.txt` is the file that the Python process can only write and the Matlab process can only read. Similarly, the file `memap_mat_to_py.txt` is the file that the Matlab process can only write and the Python process can only read. This explanation is illustrated on Figure 3.15.

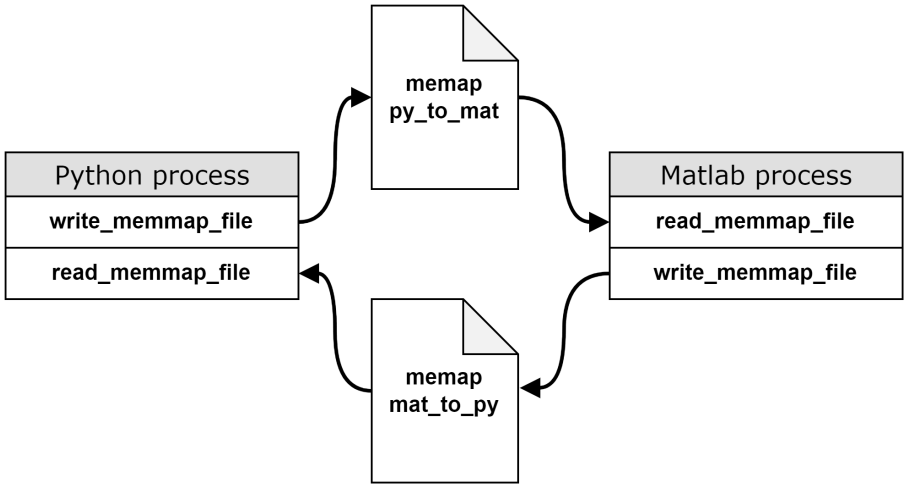


Figure 3.15: Illustration of the memory-mapped files used for passing data between processes.

The Python process is the main process that will spawn the Matlab simulation as a sub-process, and in order to handle the data passing as well as synchronization, a simple synchronization algorithm is used in order to guarantee that there are no race conditions.

Initially, the Python process creates both the `memmap_py_to_mat.txt` and `memap_mat_to_py.txt` files used for data passing. Given that most of the data that will be passed between processes are float variables, both files treat every address space as a float. This means that in order to correctly read variables on the file, the offset for reading each variable is 4 bytes.

The first address (offset 0 bytes) of both files is utilized for a synchronization variable `sync_var`, which is used to allow for a simple synchronization algorithm. The second address (offset 4 bytes) of `memmap_py_to_mat.txt` is used as a control variable for the simulation, used in case the simulation needs to be terminated or reset. The second address of the other file, `memap_mat_to_py.txt`, is used as a status variable to inform about the states of the simulation - where the simulation has ended or if it currently is on a discharge, charge, or resting state.

The rest of the address spaces are used to transfer simulation and balancing related data. In the case of file `memmap_py_to_mat.txt` the rest of the address spaces is utilised to send the balancing feedback for each cell in the simulation. Finally, for file `memap_mat_to_py.txt` the address spaces are used to send the SOC and SOH values

of each simulated cell. Refer to Figure 3.16 for an illustration of the memory address space organization of the memory-mapped files.

memmap_py_to_mat	
Addr	Data
0	sync_var
1	control_var
2	balance_control_cell_1
...	...
2+N	balance_control_cell_N

memmap_mat_to_py	
Addr	Data
0	sync_var
1	status_var
2	SOC_cell_1
...	...
2+N	SOC_cell_N
2+N+1	SOH_cell_1
...	...
2+2N	SOH_cell_N

Figure 3.16: Organization of the address spaces of the memory-mapped files.

The Python process is the one that creates the files for memory-map, does some initial setups, and then spawns the Matlab process. When it creates the files, it writes all the address spaces of `memmap_py_to_mat.txt` and `memap_mat_to_py.txt` with 0. Then, it memory-maps both files. After that, it spawns the Matlab process as an executable and passes the necessary parameters. After this, the Python process will remain in a busy-waiting loop until the Matlab process updates `sync_var` on `memap_mat_to_py.txt`.

The Matlab process on initialization will memory-map the files for data transfer and start with the simulation. The Matlab process will execute the first 30 simulation cycles and write the corresponding simulated data to `SOC_Cell_[:]` and `SOH_Cell_[:]`, update `status_var` and finally update `sync_var` on file `memap_mat_to_py.txt` to let the Python process know that new data is available. After this, the Matlab simulation will remain in a busy-waiting loop until the Python process updates it `sync_var` on `memmap_py_to_mat.txt`.

When the Python process reads `sync_var` and detects an update, the busy waiting loop ends and reads `status_var`, `SOC_Cell_[:]` and `SOH_Cell_[:]`. Now, the Python process can write data to `control_var`, `balance_control_cell_[:]` and finally to `sync_var`. After that, it will re-enter the busy-waiting loop and the Matlab process will repeat the steps, until the simulation ends.

3.7 Running the Cell Simulation Environment

The library `SimCellPack.py` is the library that handles spawning the Matlab simulation executable, which simulates the cells. This library was designed to start the simulation, control it, and send data to allow to balance the simulated cells. It uses the `mmapTx.py` library to create the necessary data transferring capabilities, and through it control the simulation. Additionally, every part of the cell pack configuration can be handled through this library. The entire source code of this thesis work can be found in the thesis github repository [39].

In order to set-up the simulation and cell parameters, the class `SimCellPack()` is called and initialized with the relevant input arguments. The arguments are listed in Table 3.7.

Table 3.7: Cell simulation parameters description

Parameter	Description	Type	Options
cellModel	The cell model can be specified, which contains all the cell model parameters needed for the simulation.	string	"P14", "E1", "E2"
numCells	The number of cells that the simulated pack will contain.	int	2-100
balancing	Selects the balancing type and balancing topology simultaneously.	string	"active", "passive"
utilization	Defines the number of simulated hours the cells will be doing discharge cycles and resting cycles. The first position in the array is for discharging time, the second position is for resting time. [discharge-time, resting-time]	[int, int]	1-100
simCycles	Number of discharge and charge cycles to run. Does not include resting cycles. A cycle occurs when the pack does one full discharge and one full charge.	int	1-100
profile	The drive cycle profile used for the discharging of the cells.	string	"us06", "udds"
seed	The seed number that randomizes the cell parameters.	int	1-65535
sampleFactor	The down-sampling rate. How many simulation steps must pass before the simulation sends data through the interface. If set to 1, it will send every data step.	int	1-60
getSOCsWhen	Controls which simulation state the simulation will send. If "all" is selected, it will send data on every state, if "discharge" is selected, it sends data only when the simulation is on the "discharging" states. Useful for training RL.	string	"all", "charge", "discharge", "dis/charge", "rest"

The `SimCellPack` class has in-built methods to start, control, communicate and terminate the simulation:

- `startSim()`
  - Starts the simulation. Sets up the memory-map interface and spawns the Matlab simulation executable and passes the configuration parameters defined on class initialization.
- `getSimStep()`
  - Gets a simulation step data if available. Must be called first before calling `sendSimFeedback()`. Blocking function.
  - Returns - [`code`, `state[]`].
    - `code`: Reports the state at which the simulation is at: 0 - simulation waiting, 1 - discharging, 2 - charging, 3 - resting, $\geq$ 4 - unassigned
    - `state[]`: Array containing SOC and SOH of the simulated cells. The first `numCells` values are SOC, the rest is SOH.
- `sendSimFeedback(feedback[])`
  - Sends balancing feedback to the simulation. The simulation will be blocked until it receives the feedback sent from this function. If $balancing =$ "active", feedback is power usage percentage. If $balancing =$ "passive" feedback is balancing current.
  - `feedback[]` - Output array of size $numCells$ that contains the balancing feedback for the simulated cells.
- `resetSim()`
  - Stops and restarts the current cell simulation. The current ongoing simulation is stopped and restarted with the initial parameters. After calling this function, `getSimStep()` must be called to get the first simulation data.
- `stopSim()`
  - Stops the simulation, kills the simulation process, and cleans data interfacing files. After calling `stopSim()`, the simulation cannot be restarted. To start a new simulation, a new simulation must be configured and started.

3.7.1 System specifications

Specifications of the hardware and software where training was done:

Table 3.8: System specifications

Specification	**Details**
Device	Laptop Computer
Operating System	Windows 10 Enterprise
Processor	3th Gen Intel(R) Core(TM) i7-13850HX 2.10 GHz
RAM	32,0 GB (31,8 GB usable)
System type	64 bit
GPU	NVIDIA RTX 5000 Ada Generation Laptop GPU
Hard Drive	500GB SSD

4

Results

4.1 Control Simulations

The next figures provide simulation runs without balancing to serve as control and comparison against simulations that include balancing, for active and passive balancing. For passive balancing, the cell pack topology differs from active balancing, and they are simulated slightly differently in the simulation. For passive balancing, cells are considered connected in series in a pack, and the current that flows through all cells is the same. However, for active balancing the cell packs are not considered connected given that the power electronics act as a connection buffer between every cell. Because of this, the current going through each cell in the pack is different and is calculated separately to emulate cells not being directly connected.

The following figures - Figure 4.1 and 4.2 - show two simulations with the same parameters, and no balancing, the only difference being the cell topology. These figures are meant to show the rate at which cells become unbalanced on each of the test topologies - one meant for passive balancing and one meant for active balancing.

Figure 4.1: Simulation of 10 cells on the passive topology with no balancing.

Figure 4.1 shows a simulation of a pack with 10 cells connected in series, the graph shows SOC over time as well as information about the details of the simulation. The pack is being discharged and charged for 6 hours as well as resting for 6 hours in a cycle, for a total of 185.3 hours. The discharge profile utilized is "udds". Cells in this pack can only be balanced during rest while they are under no load, and balancing can only be passive.

The second graph in the figure shows the difference between the maximum SOC of the pack against the minimum SOC of the pack at any given time. The maximum and minimum SOCs can change over time and might belong to different cells.

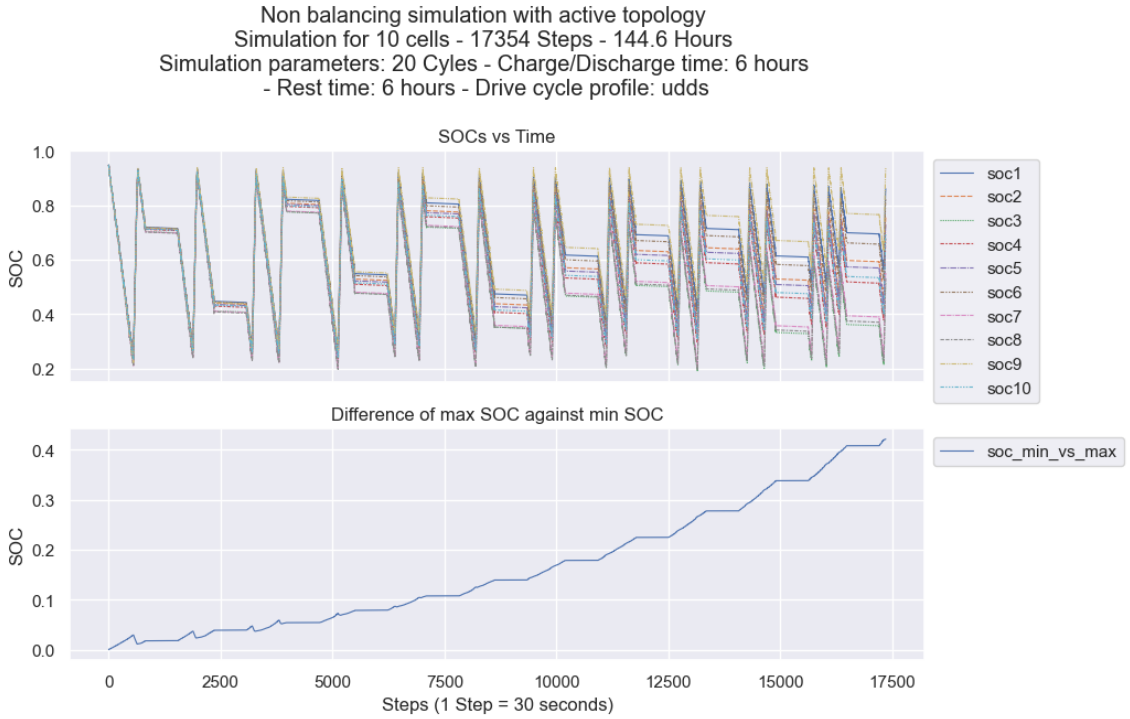


Figure 4.2: Simulation of 10 cells on the active topology with no balancing.

Figure 4.2 shows a simulation of a pack with 10 cells using the active balancing topology proposed in this work. Cells are not connected directly to one another, so they can exhibit different behaviors when compared to the cell pack shown in Figure 4.1.

The SOCs of the cells under the passive topology tend to unbalance much less over time compared to the cells under the active topology as shown on the second graph of Figures 4.1 and 4.2. For the passive topology cells, the difference between maximum and minimum SOC at the end of the simulation reached around a 0.07 SOC difference, while for the active topology, the difference reached around 0.4 SOC.

The passive topology cells seem to naturally balance during the charging phases, while the active topology shows no natural balancing tendency throughout. This behavior appears consistent for both topologies when simulations are done with varying cell parameters.

The large unbalancing behavior of the cells is a result of cells no longer being directly

connected in series to each other, which means that current flow through each cell will be much different while being discharged or charged. The currents through cells, while connected in series, have to be the same based on circuit theory, but when considered separate, no mechanism causes the cell's current to be the same as the others.

In general, cells on the passive topology tend to unbalance at a slower rate compared to the active topology in the simulation environment used in this work.

To obtain the rate at which cells unbalance on the active simulation compared to the passive simulation, we use Formula (4.1).

$$unbalance = \frac{\sum_{j=0}^{N}(SOC_{passive,j}(t) - min(SOC_{passive}(t)))}{\sum_{j=0}^{N}(SOC_{active,j}(t) - min(SOC_{active}(t)))} \tag{4.1}$$

where $unbalance$ is the ratio of how unbalanced cells, N is the number of cells in the pack and $SOC_{passive,j}(t)$ and $SOC_{active,j}(t)$ are the SOCs of the passive and active simulation respectively. We utilize this formula on the last SOC values of the simulation at the same time t to compare the rates.

Testing the formula on several different simulations with a different `seed` value, the results ranged between 6 and 9 unbalanced ratio. This means that cells on the active simulation unbalance 6 to 9 times faster compared to the passive simulation.

4.2 SOC only Active Balancing

This section shows how the active balancing approach compares to a standard passive balancing approach with SOC being the only parameter for balancing. Additionally, we analyze the SOC active balancing approach. Active balancing is done through the RL model and passive balancing is a simple SOC balancing algorithm that discharges all cells in the pack to match that of the lowest SOC cell.

The simulations used for RL model training in this section utilize the parameters shown in Table 4.1. The testing parameters remain the same except `utilization` and `profile`.

Table 4.1: Simulation training parameters for SOC only balancing.

Parameter	**Value**
cellModel	"P14"
numCells	10
balancing	"active"
utilization	[10, 2]
simCycles	50
profile	"us06"
seed	1
sampleFactor	30

Figure 4.3 shows a balancing simulation for passive balancing, to serve as a comparison point for the active balancing model.

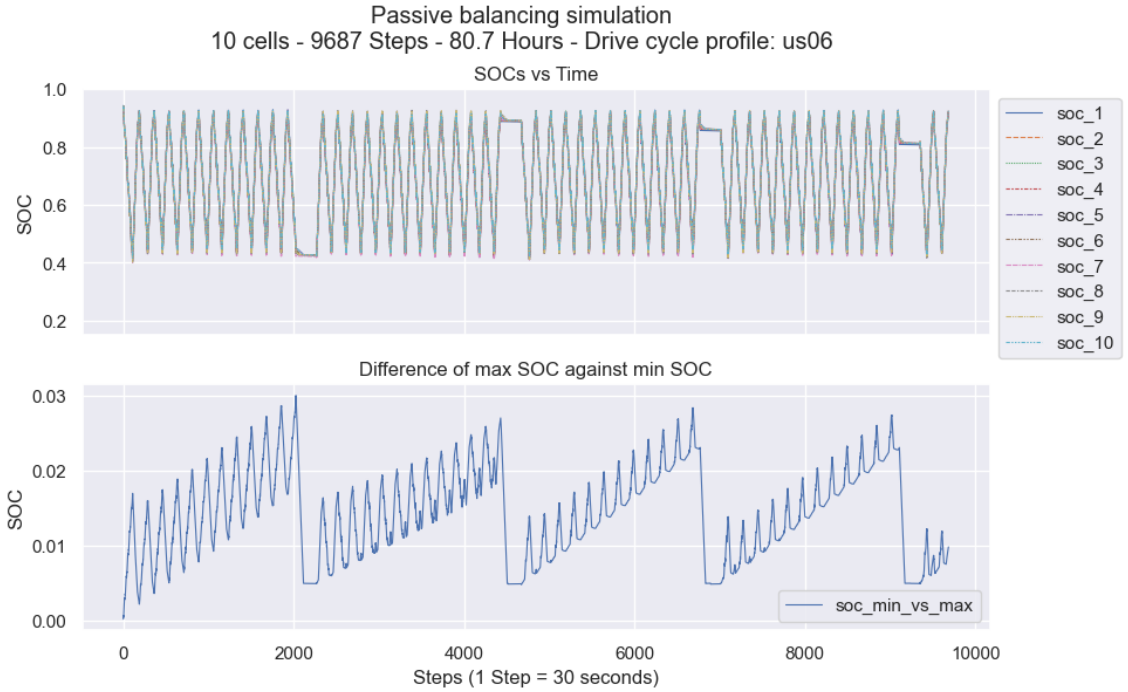


Figure 4.3: Passive balancing simulation of 10 cells - utilization = [10, 2].

The balancing actions used for passive balancing are described by Equation (3.29), and it applied only during a resting phase while the SOC difference of any cell is not less than 0.5%, as shown by the following formula:

$$i_{balance,j} = \begin{cases} 0.5\,\text{A}, & SOC_j - min(SOC) > 0.5\% \\ 0\,\text{A}, & \text{otherwise} \end{cases} \tag{4.2}$$

where for cell j, $i_{balance,j}$ is the balancing current of the cell, SOC_j is the SOC of the cell and $min(SOC)$ is the minimum SOC of the pack.

From the passive balancing simulation in Figure 4.3, we can observe that the difference of SOC starts increasing during charging/discharging cycles, reaching a maximum of 0.03 SOC difference - equivalent to 3%. The simulation with this balancing method was able to run for 80.7 simulated hours under the current simulation parameters. For this simulation, all passive balancing actions drain cells and reduce their SOCs to the cell with the minimum SOC.

Figure 4.4 shows a close-up of Figure 4.3 when the simulation enters a resting state. The balancing action of passive balancing during a resting phase causes SOC loss to achieve balance.

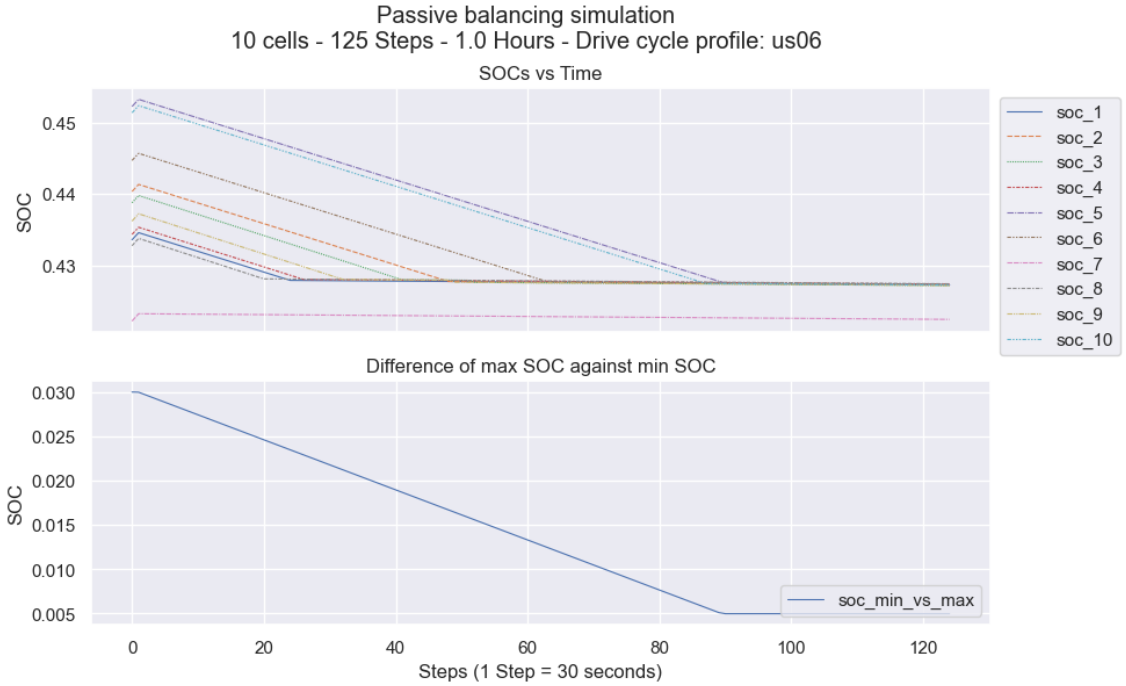


Figure 4.4: Close-up of 4.3 of steps 2025 to 2150.

When SOCs reach their highest point in Figure 4.4, the resting phase begins. SOCs are balanced to within 0.5% of the minimum SOC, resulting in a cumulative loss of 0.119 SOC between all balanced cells, and an average of 0.0132 SOC loss between all balanced cells, which equates to a loss of 1.32% SOC average. This results in a total loss of 1.729 Ah between all cells on a 10-cell pack after 17.62 hours of battery usage.

On a simulation test where `simCycles` was set to 1000, with a total simulated time of 1616 hours and `utilization` set to [10, 2], the total SOC lost to balancing was gathered. The cumulative SOC loss values after balancing range between 0.0892 SOC to 0.1379 SOC with an average SOC loss of 0.0956 SOC. On average passive balancing results in the total loss of 1.389 Ah for the pack after 17.62 hours of battery utilization. Over the 1616 hours of simulation time, the total charge loss resulted in 113.91 Ah. When the test is run with the active balancing RL model, the simulation lasts for 1576 hours.

The same 1000 `simCycles` test was repeated but with `utilization` set to [20, 2]. This time, the simulated time was 1534 hours. The cumulative SOC loss to passive balancing ranged between 0.1463 SOC to 0.2319 SOC with an average of 0.1907 SOC. On average, 2.77 Ah of pack charge was lost to passive balancing on 35 hours of battery utilization. Over the 1534 hours of simulation time, the total charge loss resulted in 113.62 Ah. When the test is run with the active balancing RL model, the simulation lasts for 1496 hours.

Figure 4.5 shows an active balancing simulation with the same parameters as the passive balancing simulation, and the balancing actor is the RL model trained for SOC-only balancing.

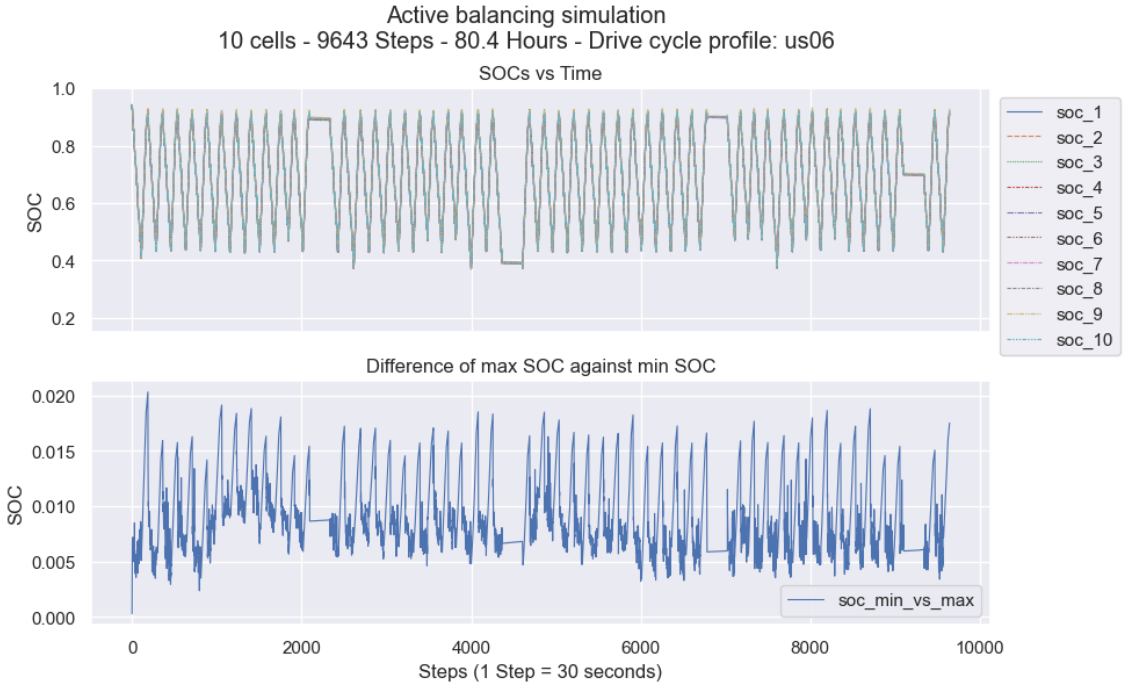


Figure 4.5: Active balancing simulation of 10 cells - utilization = [10,2].

The simulation from Figure 4.5 shows how during discharge phases the maximum SOC stays balanced in a range between 0.3% and 2% SOC to the lowest SOC. Balancing remains consistent throughout the simulation, reaching a simulated time of 80.4 hours, which is less time compared to the passive balancing simulation. However, cells on the balancing topology have a much quicker unbalancing rate compared to the passive topology version, meaning that even when cells unbalance quicker, the active balancing topology can generate acceptable balancing results on a partially trained model with a cell pack system that unbalances 6 to 9 times faster.

During the simulation, we can observe peaks on the SOC difference plot, and what happens on a cycle is difficult to see. Figure 4.6 shows a close-up of a charging, discharging, and resting cycle of a similar simulation.

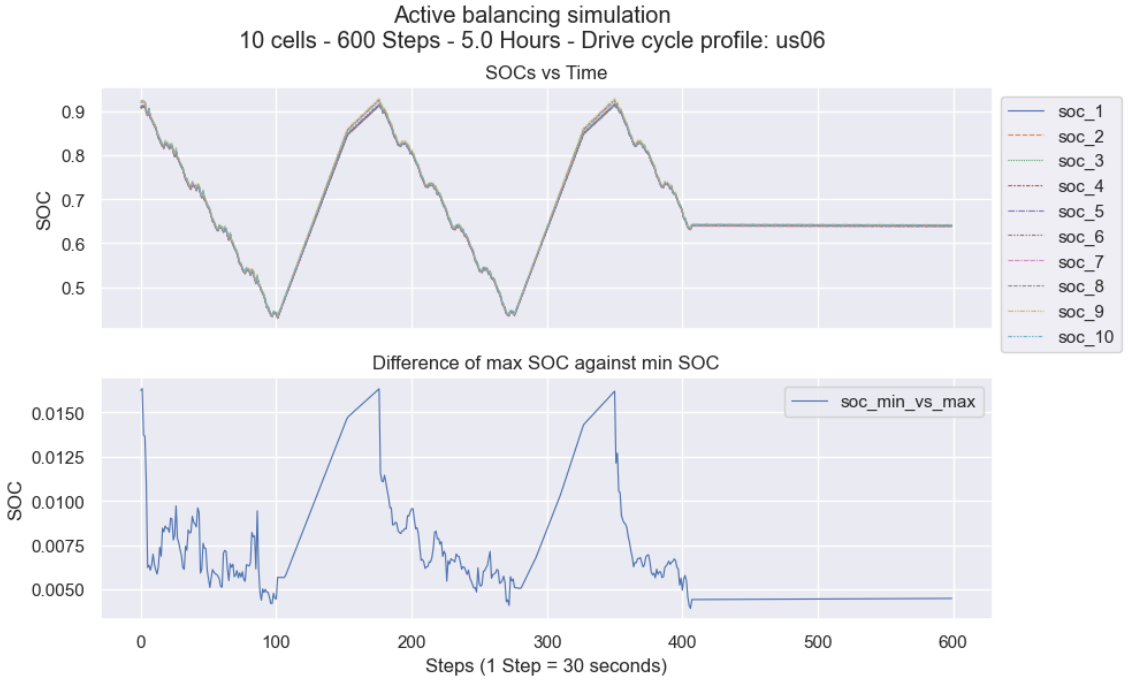


Figure 4.6: Active balancing simulation of 10 cells - close-up.

As previously shown in Section 4.1 Control Simulations, the active topology simulation unbalances constantly, this results in the peaks we observed in the previous figure. During discharge, the model manages to balance SOCs to less than 1% difference. When a charging phase ends, cells reach around 1.6% SOC difference, meaning they gain 1% SOC difference in a single charging event. By factoring out the unbalancing behavior of cells during charging, cell SOC would consistently remain on a 0.3% to 1% SOC difference during discharge, without the disadvantage of charge loss as opposed to passive balancing.

4.3 Active SOC and SOH Balancing

In this section, cell SOH is introduced into the RL model training. We explore the results on how the model balances SOC and SOH on different SOC ranges as well as explore the possibilities and limitations of this balancing approach in this work.

The reward of the models is described in Equation (3.40), we test 4 trained models with both SOC and SOH balancing where the reward thresholds used are 1%, 2.5%, 3.5% and 5%.

The RL models used in this analysis were trained with the following simulation parameters:

Table 4.2: Simulation parameters of the trained SOC and SOH balancing models.

Parameter	**Value**
cellModel	"P14"
numCells	10
balancing	"active"
utilization	[2, 2]
simCycles	10
profile	"us06"
seed	1
sampleFactor	30

The RL models were trained on the drive cycle profile "us06" with `utilization` as [2,2] due to the drive cycle profile causing cells to discharge quicker increasing cell unbalance. Also, the cell discharge and resting cycle is 2 hours for discharging and 2 hours for charging.

The testing parameters for result gathering were done with the following parameters:

Table 4.3: Simulation parameters for testing SOC and SOH balancing models.

Parameter	**Value**
cellModel	"P14"
numCells	10
balancing	"active"
utilization	[20, 2]
simCycles	10
profile	"udds"
seed	1
sampleFactor	30

For testing, we use the drive cycle profile "udds" which discharges cells slower, which makes the simulations run for longer, making the RL model have to balance for increased periods. Also, `utilization` is 20 hours discharge and 2 hours resting to have less balancing downtime, to allow the model to balance for the most time possible and analyze their behavior.

For comparison against future result examples, Figure 4.7 shows an RL model where the threshold is set to 0%, meaning that it only does SOC balancing.

The simulation manages to keep the SOC balanced between 0.5% and 1.5% during the discharge phases, netting a total simulation time of 55.1 hours. Since this model is not trained to balance SOH - even though it is present in its observation space - the SOH values very quickly unbalance from the start and keep unbalancing until the simulation stops.

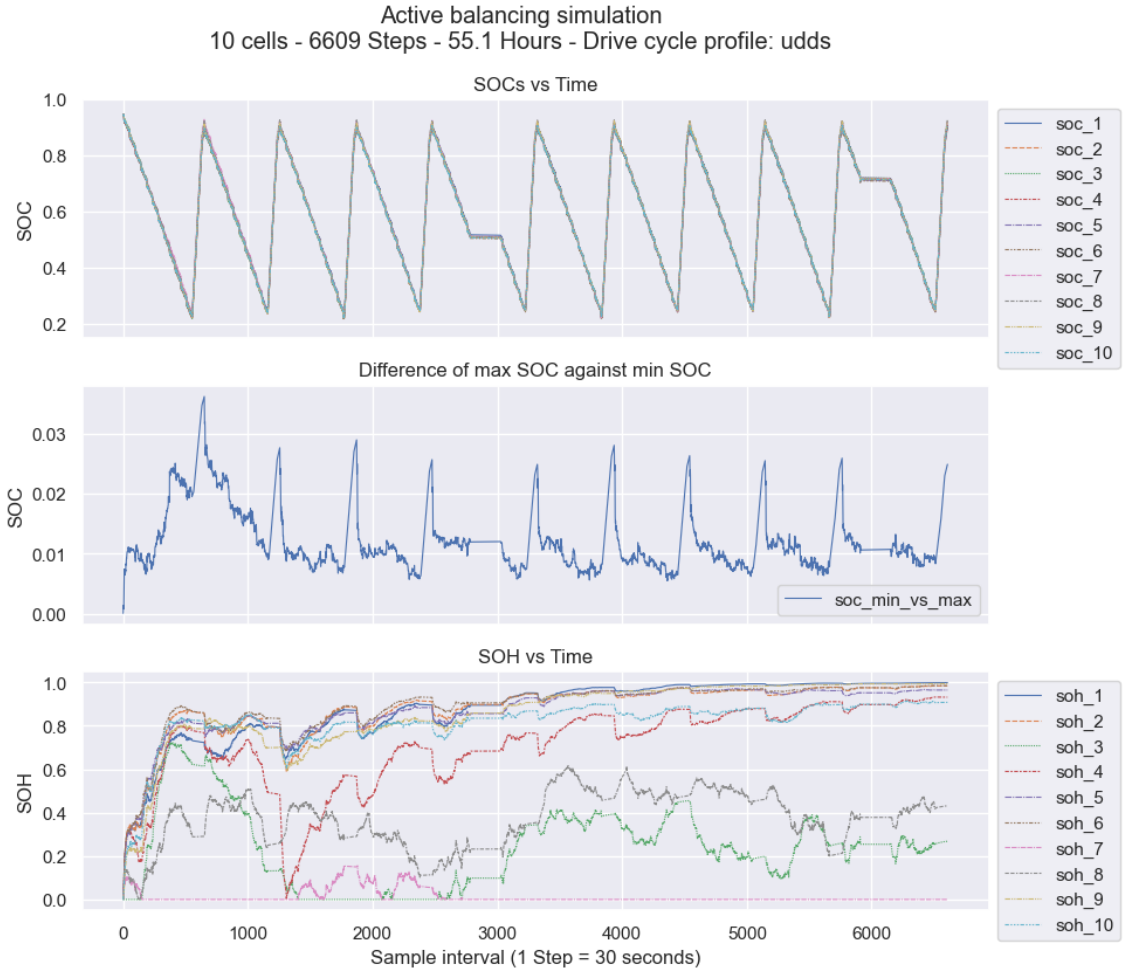
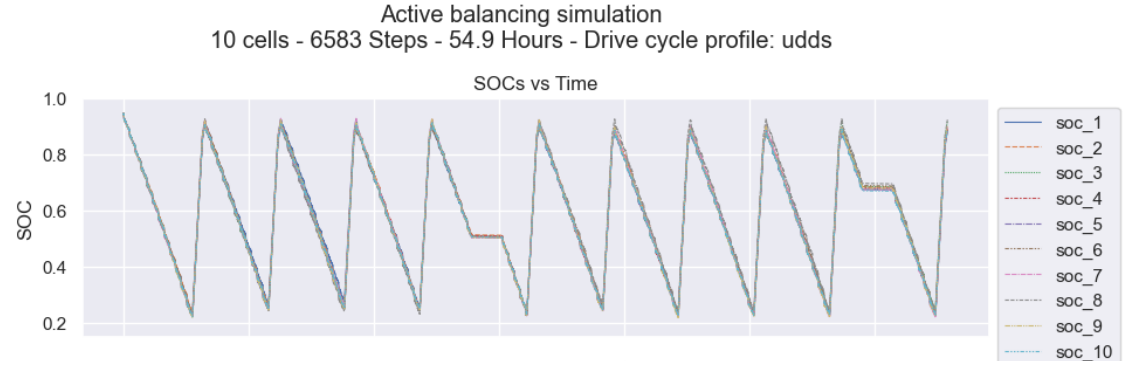


Figure 4.7: Active balancing for SOC only (0% threshold)

4.3.1 SOC and SOH Balancing with 1% Threshold

Figure 4.7 shows a 10-cell simulation where the RL model was trained to balance SOC and SOH with a threshold of 1% for the reward.



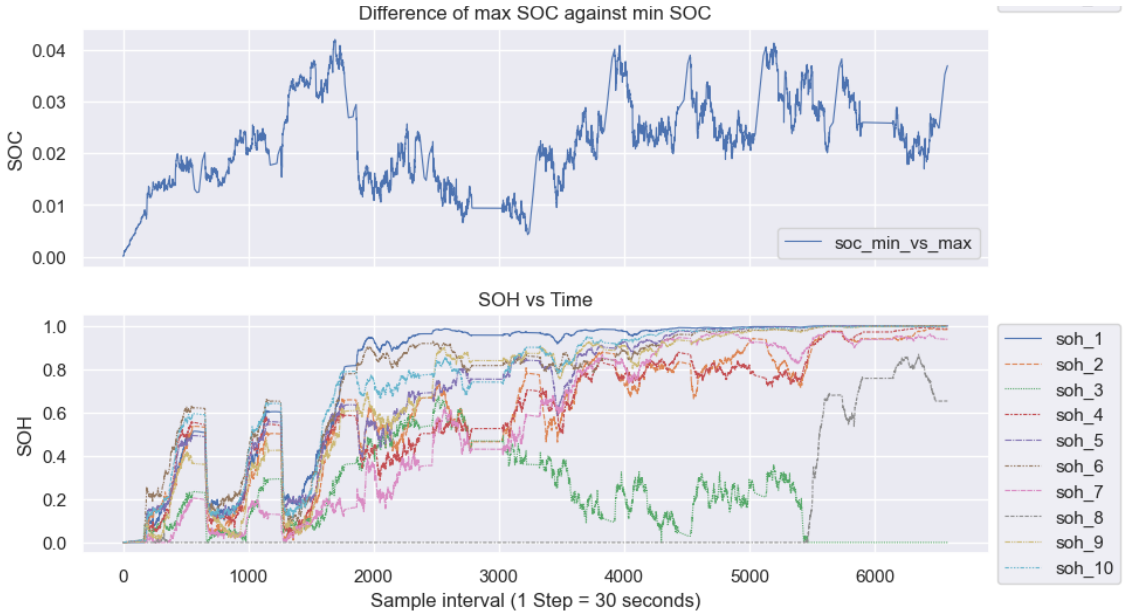


Figure 4.7: Active balancing for SOC and SOH - 1% threshold - case 1.

When observing the second graph, we can see that the SOC difference remains below the 0.04 SOC mark when the expected result is for SOC to be at or below the 0.01 SOC mark, which is the 1% threshold line. The model can balance decently even though not being able to strictly adhere to the set threshold for SOC balancing.

When considering SOH at the same time, we can observe from the third graph that SOH is not getting well balanced, when the expectation for SOH balancing is that they remain as evenly distributed as possible. This means that the SOH values of the different cells should be close together. Cell 1 appears to be consistently experiencing the most power dissipation while Cell 7 is not close to sharing the same throughput as even the lowest demanded cells. This, in turn, means that Cell 1 is experiencing the most throughput, which can potentially cause it to age at a faster rate than the rest.

Due to the threshold of 1% SOC balancing being very small, the model does not appear capable of managing good SOH balancing in tandem with SOC on all cells. Not only are the cells not achieving 1% SOC balance, but the model still struggles to perform the double balancing efforts on all cells. This can suggest that such restraints on the reward and balancing may not be compatible, as they could be interacting in opposite directions.

However, the majority of cells at higher SOH levels are relatively close to each other and stay relatively balanced. This could be considered as partial balancing of the cells, and even if all cell usage cannot be balanced over time, some amount of balancing could also prove potentially useful.

Figure 4.8 shows another simulation case.

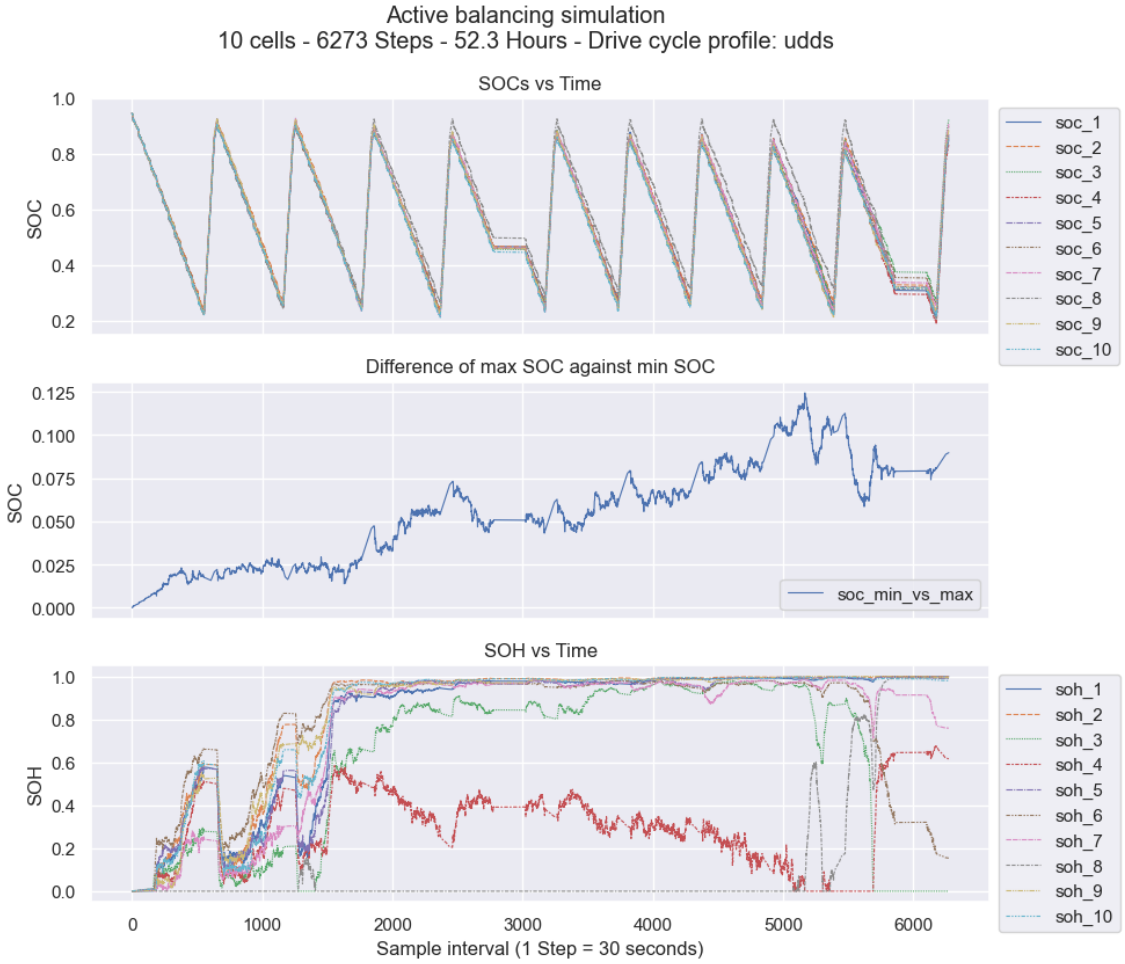


Figure 4.8: Active balancing for SOC and SOH - 1% threshold - case 2.

In this simulation example, we can observe that the model is not capable of balancing SOC correctly, given that over time the cells are becoming increasingly unbalanced, with the SOC difference reaching above 0.1 SOC, meaning that the cells are unbalanced to around 10%, far from the balanced range. If we observe SOH, after a while SOH levels start to diverge greatly and the model is not capable of keeping them in closeness. Similar to the results shown in the first simulation.

Additionally, when SOC balancing is not done effectively, it directly impacts the amount of time that the simulation runs, due to the wasted potential. This is confirmed by the simulation runtime. In case 1, cells remained balanced to within a maximum range of 4%, and 54.9 hours of simulated time was achieved. In the second case, cells became too unbalanced, resulting in a simulated time of 52.3 hours.

Figure 4.9 presents a third simulation case.

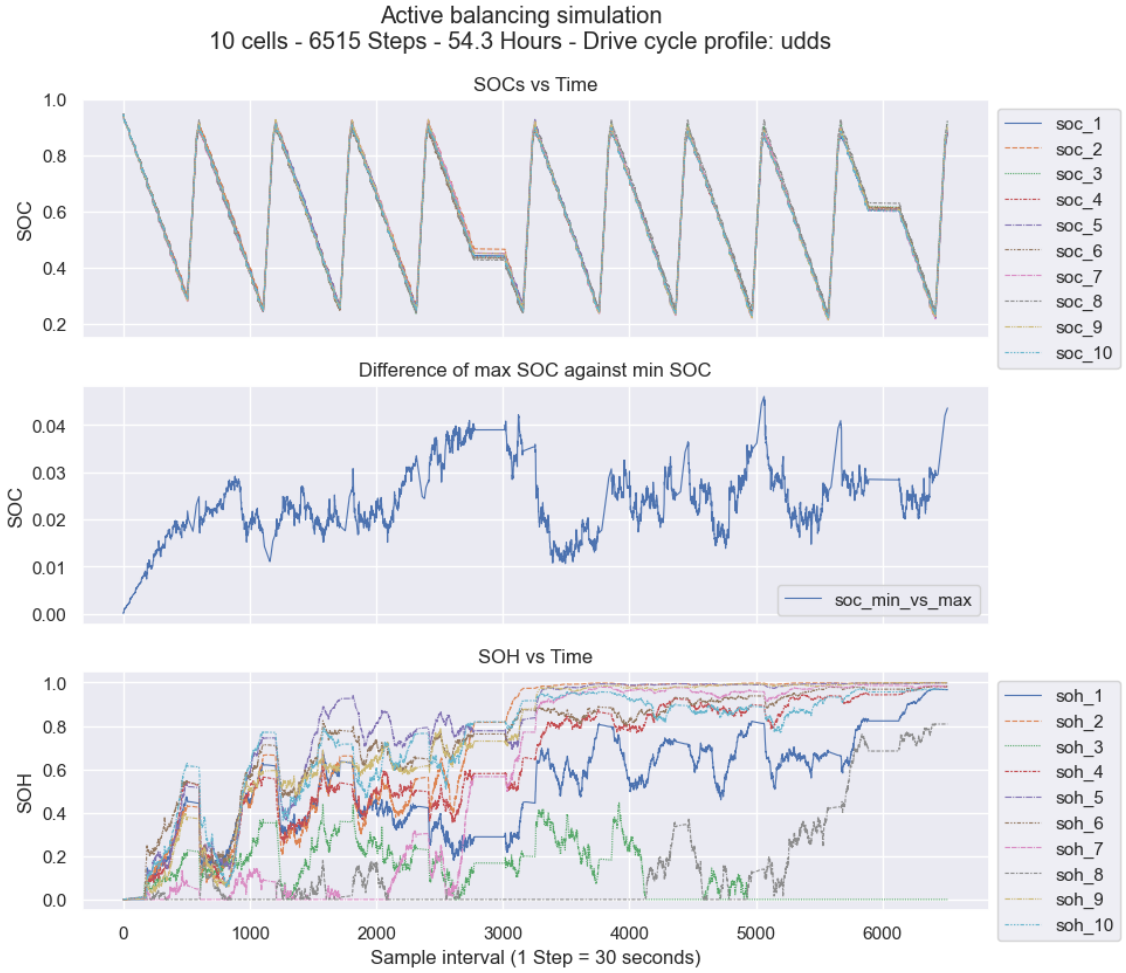


Figure 4.9: Active balancing for SOC and SOH - 1% threshold - case 3.

Similarly to the first case even, though cells are staying in an acceptable balancing range between 1.5% and 4%, the model is not capable of maintaining SOH balance initially and later achieving partial balance of a reduced group of cells. For each case, the simulation time as well as SOC balancing we inconsistent and the RL models were not able to show a satisfactory balancing result overall.

4.3.2 SOC and SOH Balancing with 2.5% Threshold

Figure 4.10 shows a 10-cell simulation where the RL model was trained to balance SOC and SOH with a threshold of 2.5% for the reward.

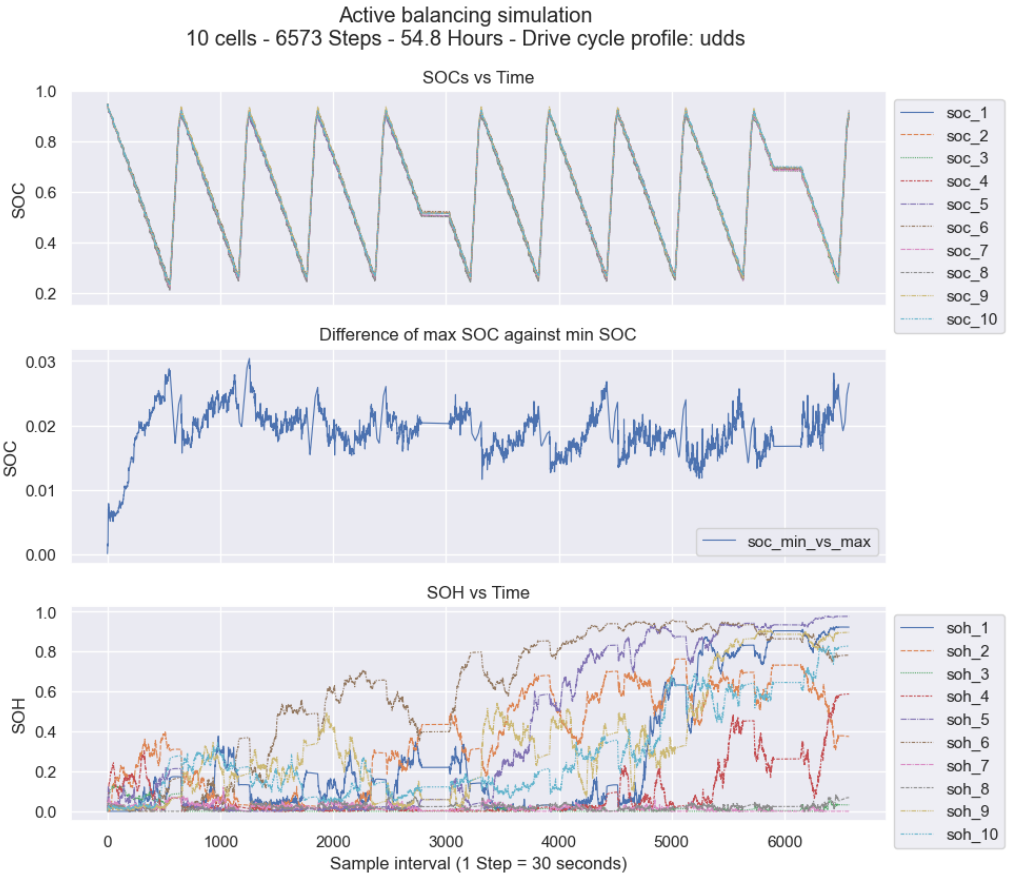
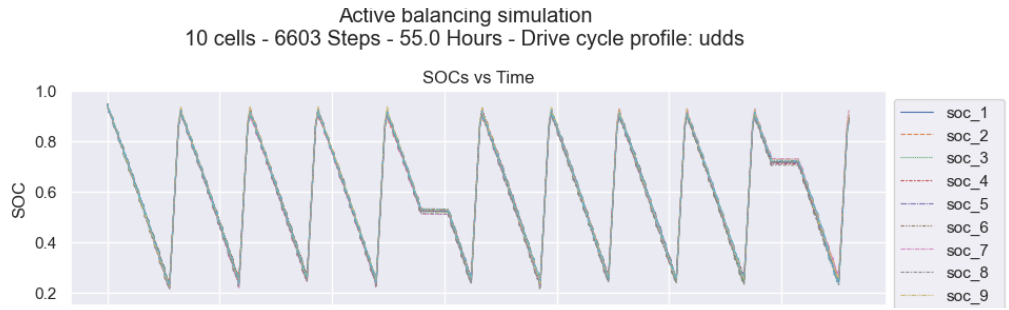


Figure 4.10: Active balancing for SOC and SOH - 2.5% threshold - case 1.

For this case, the SOC of the cells stays balanced around the expected threshold, keeping a range between 1% and 3% for the worst case, mostly hovering around 2% SOC difference, the SOC can be considered well balanced throughout the simulation duration. For SOH balancing, values start relatively close and stay like that for a duration until eventually, values spread out and SOH is no longer well balanced.

Given that this RL model has learned to balance SOC correctly, and keep the SOCs in a small range, that same range is the space where the model is allowed to perform SOH balancing more effectively. Even though the model is incentivized via the reward to keep cells under the 2.5% range of difference, managing SOH balancing in that small range may not be enough space to allow for an effective SOH balance throughout all the cells.

Figure 4.11 illustrates another case:



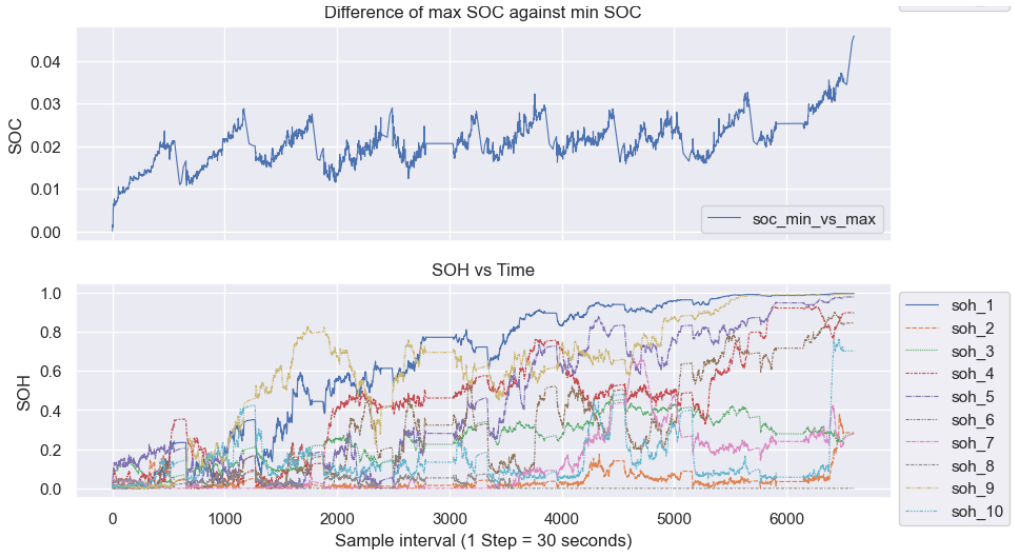
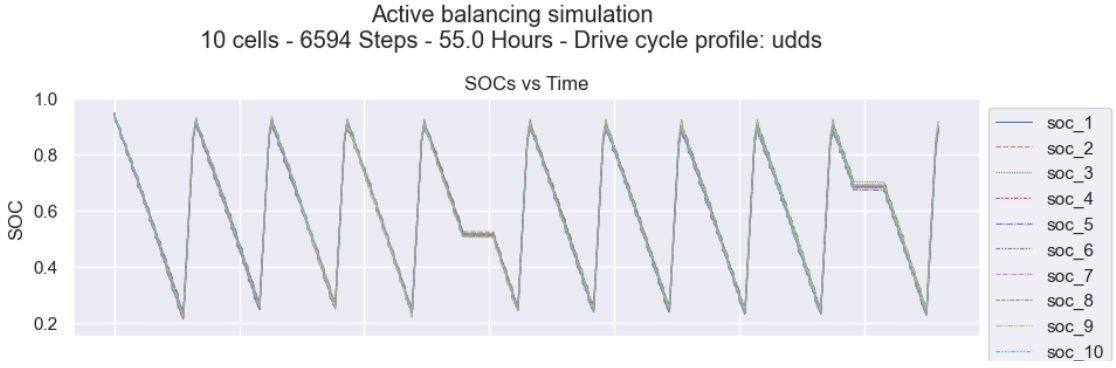


Figure 4.11: Active balancing for SOC and SOH - 2.5% threshold - case 2.

For the second case, we can observe that the results are similar to the first case. The simulation managed to run for slightly longer, not a substantial difference. In terms of SOC balancing, the model is still performing as expected, keeping SOCs between 1% and 3% for the most part. At the end of the simulation, we can see that SOC starts to reach above 4% SOC difference, this may be partly due to the limitations on which the model was trained on.

SOH balancing shows a clear difference when compared with no SOH balancing in Figure 4.7, the model manages for a short time to keep SOH balance initially, but later cannot control the ever-increasing unbalancing events.

Figure 4.12 illustrates yet another case for 2.5% threshold:



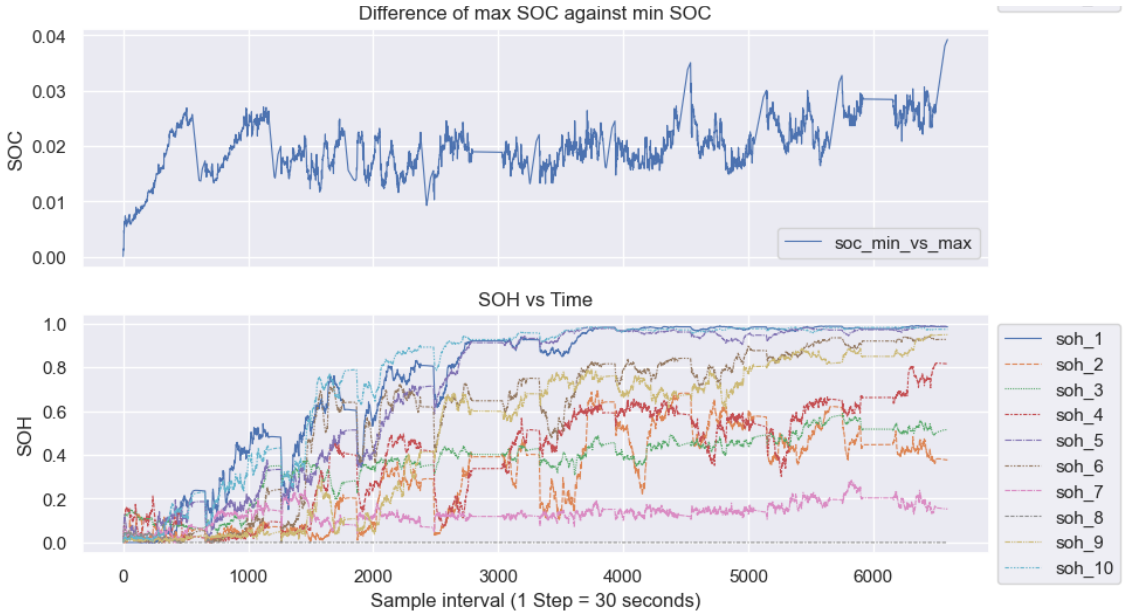


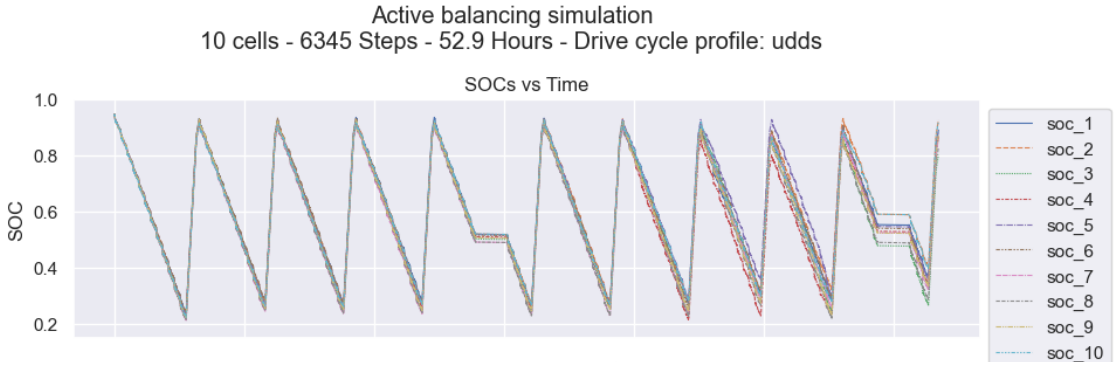
Figure 4.12: Active balancing for SOC and SOH - 2.5% threshold - case 3.

In this third case, we observe a very similar result to the previous cases, where SOC remains balanced in a small range while SOH does not show improvement after some time has passed.

For all shown cases, compared to the RL model with a 1% threshold, SOC balancing is much more consistent, while SOH balancing shows a slight improvement. Additionally, the graphs for SOH at the initial portions of the simulations do not disperse as quickly, and this trend is explored when testing for 3.5% and 5% thresholds. However, a threshold of 2.5% may also not be enough range to allow for improvement in SOH balancing.

4.3.3 SOC and SOH Balancing with 3.5% Threshold

Figures 4.13, 4.14 and 4.15 illustrate cases 1, 2, and 3 respectively for simulations with 3.5% threshold.



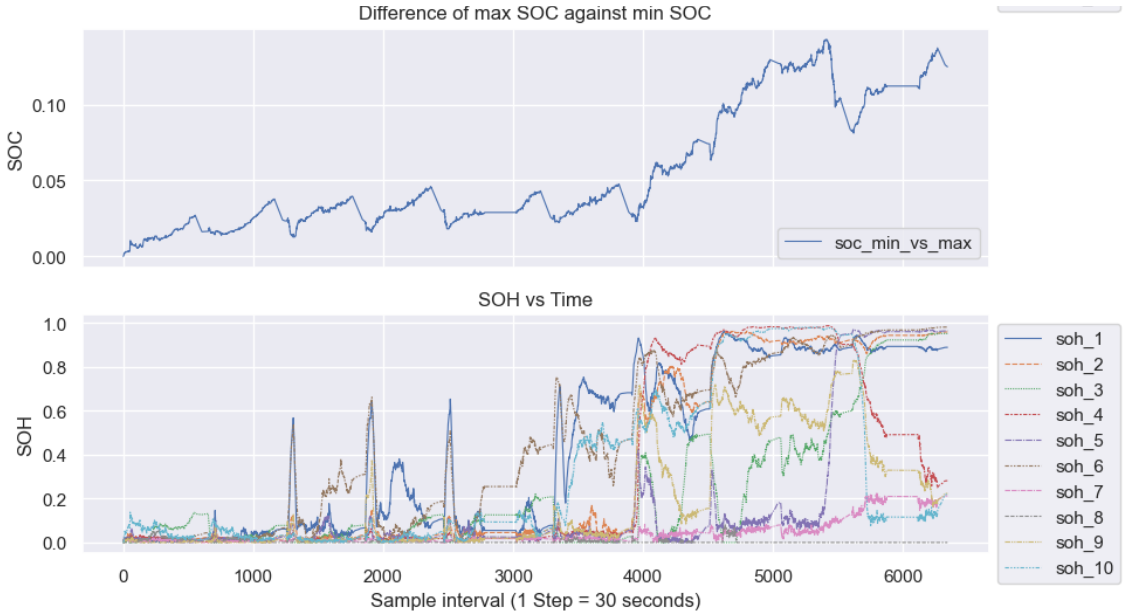


Figure 4.13: Active balancing for SOC and SOH - 3.5% threshold - case 1.

Figure 4.14: Active balancing for SOC and SOH - 3.5% threshold - case 2.

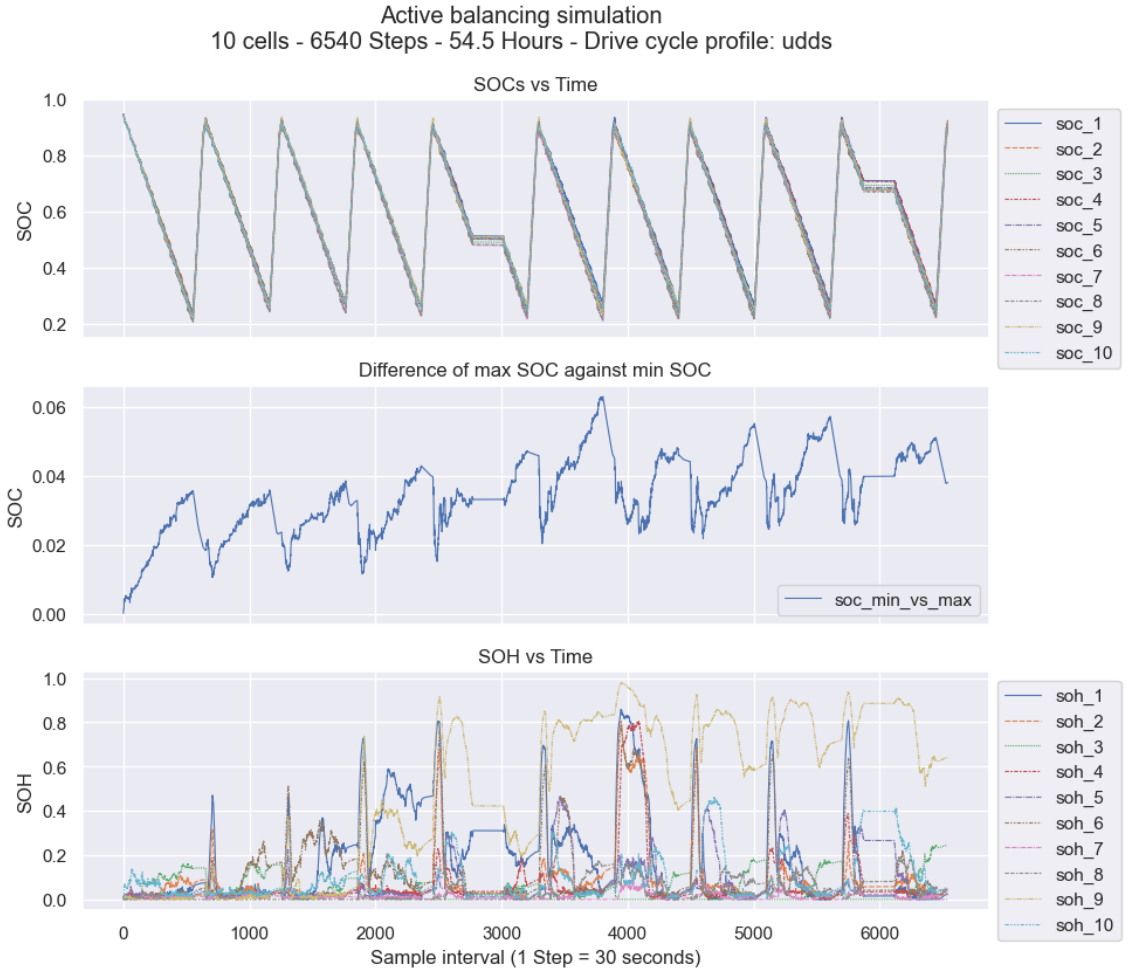


Figure 4.15: Active balancing for SOC and SOH - 3.5% threshold - case 3.

For the first case we can observe that around 4000 steps, the SOC balancing starts exceeding the balancing ranges set for this model. This is the case of a simulation that if run for longer would have failed to balance SOC. This is also an indication that the model was not fully able to learn the desired policy for SOC balancing on longer simulation times, which is expected.

However, cases 2 and 3 manage to balance SOC much closer to the threshold they were trained on. For cases 2 and 3, for the first 3000 simulation steps, the SOC balancing was able to be maintained between 2% and 4% SOC difference. Additionally, SOH balancing shows better results. Case 3 manages to keep most SOH levels in proximity for longer periods, and as some SOH values start to unbalance over time, the model manages to reduce their difference as long as the simulation continues, allowing the model to balance throughout the simulation.

By allowing the RL model more SOC space to be able to account for SOH balancing, we can see that the model starts showing better results. However, by increasing the SOC threshold we reduce how tightly the SOC balance is kept, this in turn sacrifices utilization time, causing the battery pack to need to be charged sooner. This occurs because charging starts when any cell reaches its lowest voltage threshold, and charging

stops when any cell reaches its maximum voltage threshold. This means that if cells are too unbalanced when a charging even happens, charging will not be as effective with unbalanced cells resulting in wasted capacity.

4.3.4 SOC and SOH Balancing with 5% Threshold

Figure 4.16 shows a balancing simulation with a 5% threshold.

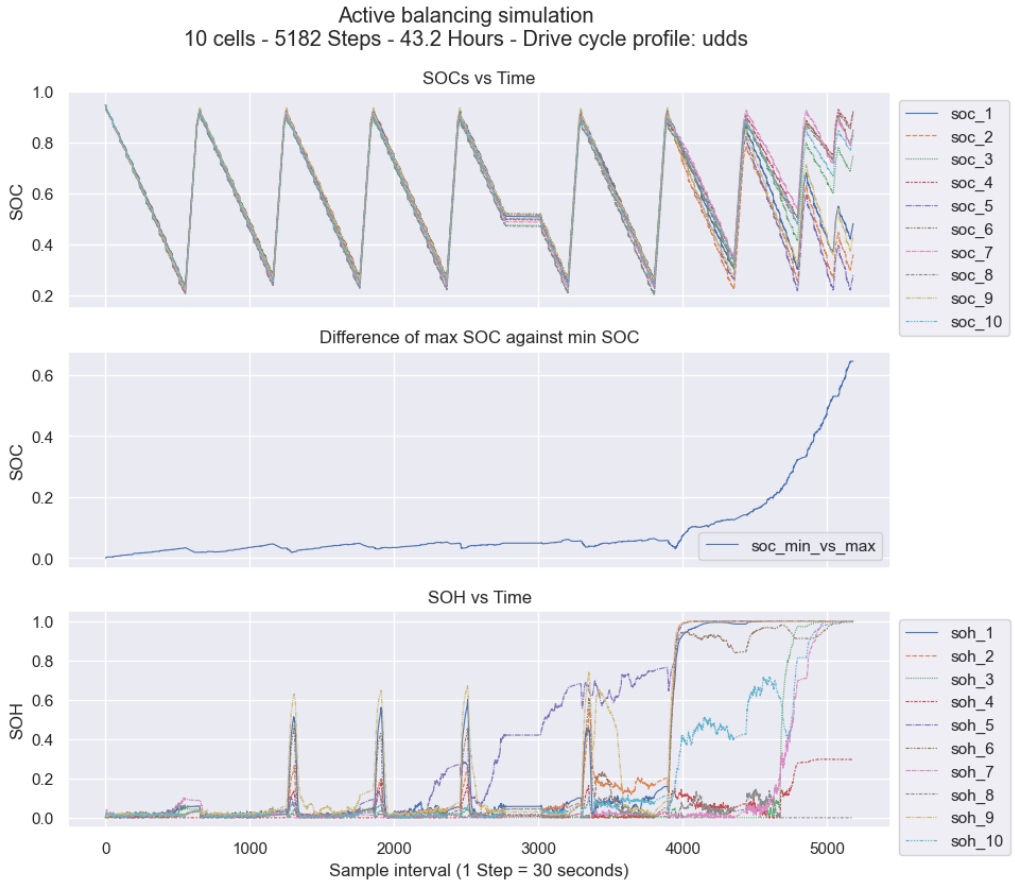


Figure 4.16: Active balancing for SOC and SOH - 5% threshold - case 1.

In this case, the RL model fails to balance at the later stages, after which the simulation ends. This is mainly due to the model not being trained for longer simulation cycles and failing to adapt, this is expected. However, we can focus on the first 4000 steps of the simulation in Figure 4.17.

On the first 4000 simulation steps, the model manages to balance SOC on a 2% to 6% range while also keeping SOH values closely balanced even after charging stages. This model has a great SOC range to allow it to balance SOH, and we can see the distinction compared to the lower threshold cases.

This suggests that to enable SOH balancing, cells need to be allowed up to a certain threshold of unbalance. Naturally, the drawback comes in the reduction of range due to temporal waste of capacity.

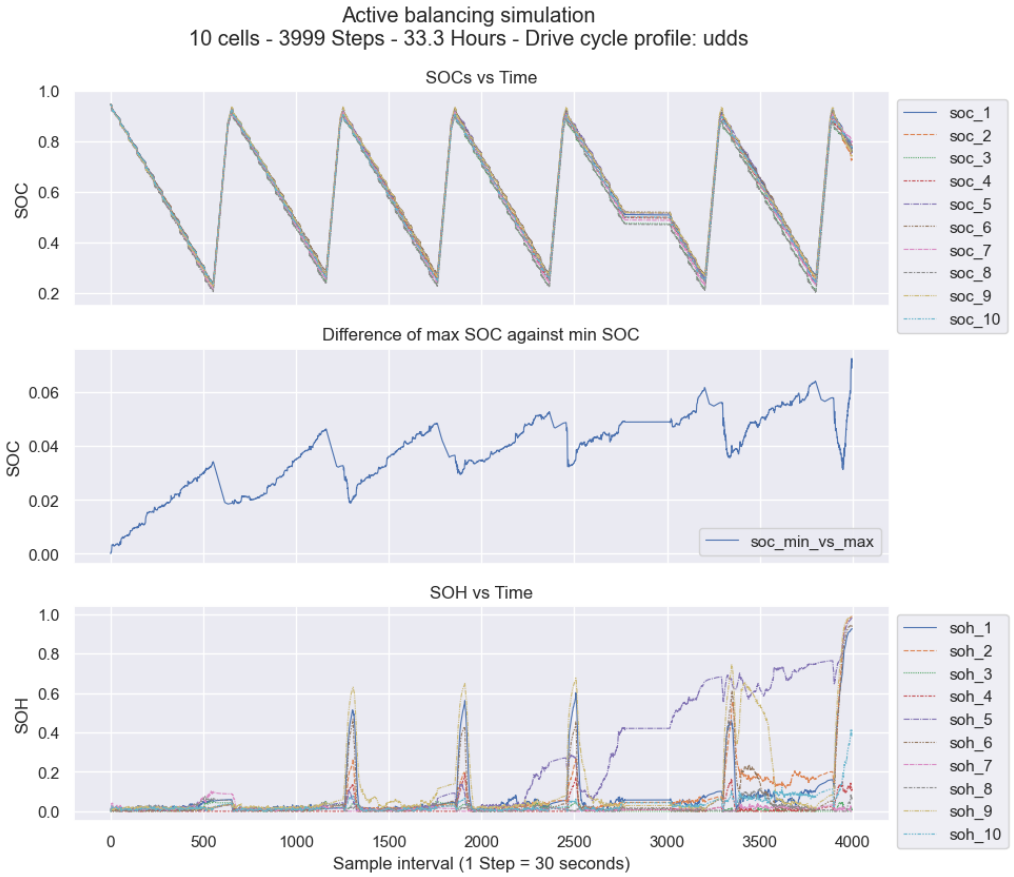


Figure 4.17: SOC and SOC 5% threshold - case 1 - First 4000 steps.

4.4 Discussion

The active balancing approach presented in this work aims to be able to balance cells in a battery with the use of RL in a manner that avoids wasting energy compared to the passive balancing alternative. Additionally, we train the RL models to not only balance SOC to keep cells under operating conditions, but SOH as well to homogenize the aging of the cell pack and extend the battery life-cycle.

The trained RL models manage to balance cell SOCs during the discharging phases between 0.3% to 2% where cells being under 2.5% SOC difference are considered balanced. The active balancing approach was able to achieve comparable runtime to passive balancing, between 0.37% to 2.47% less simulation time on a system where cells unbalance at a rate 6 to 9 times faster. This test shows that the active balancing approach was capable of making effective use of the cells in a pack that would not be suitable in a passive balancing system. The loss to balancing for passive balancing simulations resulted in between 0.0704 Ah to 0.074 Ah every hour the cell pack is under usage which equates to losing 0.484% to 0.509% of a single cell's capacity to balancing every hour. On a system where cells unbalance approximately 6 to 9 times faster, it would correspond to losing around 3% to 4.6% of a single cell's capacity to balance each hour of battery usage, based on the cell parameters used for these simulations.

We show that the active balancing approach works effectively at balancing cells with high unbalancing rates. However, active balancing can only make up for the difference that passive balancing makes, which suggests that on systems that have very low unbalancing tendencies and the wasted energy to balancing is small, active balancing may be inadequate. Nevertheless, active balancing may be an ideal solution for systems where cells are non-uniform to an extent, allowing to potentially extend the operational effectiveness of cells in a pack.

Complementary to cell active balancing during discharge, implementation of active balancing during charging may also show potential and synergize balancing in an increasingly optimized way. Balancing during charging reduces the wasted capacity of unbalanced cells, which could help reduce 1% SOC unbalance on active balancing simulation charging phases, allowing more cells to reach their maximum SOC and provide charge for a longer time.

Due to the active balancing RL models only being trained during discharge phases, they are blind to how SOCs behave during the charging and resting phases. This poses a challenge for the training of the RL models, given that it is difficult to allow a single model to learn what occurs during charging without being able to interact during that phase. A fully trained model would expect that the model intentionally unbalances cells during discharge in a way that it will naturally balance again during charging. However, this may not be feasible for a single mode, as it would likely increase its complexity. A potential solution may be having a model that can predict cell SOC after charging and work in conjunction with the active balancing model. This would discard the need for balancing during charging, yet the implications of this need to be further studied.

When combining SOC balancing and SOH balancing, the models required an increasing difference between cell SOCs to be able to also account for SOH balancing. The SOC and SOH models trained for 1% and 2.5% threshold, although capable of balancing SOC between 1% and 3%, were incapable of balancing SOH simultaneously.

The 3.5% threshold SOC-SOH model was able to keep cell SOCs in a range between 1% and 6% while simultaneously handling SOH balance in the 40% range for the majority of cells. This indicates that cells were maintaining SOC balance as well as being aged at similar rates within a 40% difference. For the case of SOC-SOH balancing with a 5% threshold, the SOCs remained balanced in a 2% to 6% SOC window while most cells' SOH was kept balanced within 20%.

The initial results of SOC-SOH balancing appear optimistic, still the trained models would fail to balance correctly after a time due to the limited capacity at which models were trained. However, increasing the threshold at which SOCs are considered balanced showed considerable improvements in SOH balancing. This suggests that SOC and the SOH metrics utilized in this work are not fully complementary and rather than being an added benefit, instead, they require careful tuning to reach an equilibrium point where both SOC and SOH balancing can be effective. In addition, SOH is a metric that aims to estimate the health of the cells, and there is no method to directly measure it. Several estimation alternatives for SOH may work adjacently or in a supportive manner so SOC balancing. In addition, as mentioned before, SOC-SOH balancing could potentially benefit from balancing during charging. Having control over how much each

cell is charged alongside a relatively large SOC balancing threshold, can potentially have the necessary capacity to allow for a much more effective SOC-SOH balancing strategy.

Finally, results on SOC and SOH balancing also suggest that maintaining a battery pack's SOH balance comes at a loss of effective battery capacity and ultimately - range. Temporarily sacrificing range to extend a battery's life span, especially on EVs where batteries comprise a large portion of the cost, could be an appealing solution. In applications where the range limits of an EV are seldom utilized, it would make sense to intentionally decrease the vehicle's range to preserve the battery for longer.

5

Conclusion

5.1 Conclusion

In this work, we investigate how RL-based cell balancing can help equalize the lifespan of lithium-ion batteries by increasing the efficiency usage of a battery pack's cells. We built upon a battery cell pack simulation application to create an RL training and testing framework focused on cell balancing. This enabled the design of a RL-based cell balancing approach based on a reconfigurable battery topology, where balancing is done during a battery's discharging phases. With this balancing approach, we investigated the viability of cell SOC balancing during discharge incorporating cell SOH balancing, allowing for the homogenizing of cell health in a pack.

We have found that an RL controller can be trained to handle the high-dimensional continuous action spaces, as well as the non-linear relationships within a battery to do effective balancing.

Simulation results of our work show that balancing during a cell pack's discharge phases can achieve a comparable runtime to that of a passively balanced pack with a lesser unbalancing rate. Active balancing during discharge managed to get within 0.37% to 2.47% of the utilization time of a passively balanced pack, on a pack with a 6 to 9 times faster unbalance rate.

Additionally, by introducing cell health into the balancing considerations we show that by reducing the strictness of cell SOC balancing, it can be possible to also balance cell health. Simulation results show that by increasing a cell pack's balancing threshold from 1% to 5% - allowing cells to become more unbalanced - cell "aging" or SOH can be balanced in a way such that each cell's energy throughput also remains similar to the other cells in the pack. Initial results show that the majority of cells in a can be balanced within 20% to 40% range for SOH, depending on the threshold configuration.

The experimentation conducted in this thesis could help open the doors to changing the way batteries are balanced, allowing for battery packs to last longer and take advantage of it resources as efficiently as possible.

Lithium-ion batteries have seen an explosion in popularity for various applications as technology never-endingly continues to advance, but the materials they are composed of are highly toxic and even more so environmentally harmful. With this research, we hope to assist into making batteries and limited resources be used efficiently and for longer.

5.2 Future Work

The simulations for the cells assume a constant temperature working condition of 25° Celsius. For our purposes we consider having constant temperature adequate, since we aim to explore the capacity and potential of the RL model applied to cell balancing. The datasheets from the cells rely on constant temperature for their measurements. To increase the fidelity of the simulation the constant temperature can be replaced for a dynamic temperature model which simulates temperature changes during battery operations, as well as changes in ambient temperature from the environment.

To simulate the cells being utilized under realistic conditions, we make use of EV simulations based on real-life car models under real-life measured driving cycles (driving through a city, highways and urban areas). These provide simulated drive cycle current and power demand profiles under which the simulated cells will be discharged. However, the battery pack cells used in the EV simulation and the cell models used in the cell simulation are not the same, given that these simulations are separate from one another. To minimize the mismatch of the cells from the vehicle models in the EV simulations and the cell models used in the cell simulations, we matched the cell model cells to the simulated vehicles which are the closes in terms of capacity. A uniform array of datasheets for pack and cell sensor data, taken from a vehicle in-operation can help model a more accurate simulation environment of the battery.

The RL models are trained based on SOC, which is an estimation given it cannot be directly measured. The effectiveness and accuracy of our proposed method hinges on the precision of the data and the SOC algorithm's specific requirements. More accurate estimation methods can increase the performance of the balancing algorithm by allowing for greater degrees of precision.

Within our RL algorithm selection a large focus was sample efficiency in training. Sample generation in online RL is bounded only by computation time. However, it becomes sample efficiency becomes critical in the case of offline RL, where the amount of samples is limited to the available datasets. Future projects can pursue conversion of the online RL training framework provided to an offline RL approach, and make use of the training techniques and algorithm observations to limit the size of the necessary dataset.

The RL models are trained on a limited simulation time, which makes them not suitable for large simulation times and they become incapable of balancing for long periods of time. This limitation was mainly due to the time it takes for training to occur on long simulation setups. Training models to achieve these tasks may require rethinking and testing new rewards, as well as having more substantial time for training. Thus, we analyze the models on a limited capacity, and attempt to bring to light the potential uses or results that may drive future research. Additionally, the simulation results are the best results that were able to be taken, given that the models could not consistently provide the same simulation results, showing that more training time, as well as training cases, are needed in order to obtain a fully trained model.

Bibliography

[1] P. Ahir, Y. Xie, and G. Happawana, "Impact of battery cell imbalance on the voltage behavior of commercial ni-mh ev/hev battery modules," in *2022 IEEE Vehicle Power and Propulsion Conference (VPPC)*, 2022, pp. 1–3. DOI: 10.1109/VPPC55846.2022.10003355.

[2] J. Li, "Economic analysis of retired batteries of electric vehicles applied to grid energy storage," *International Journal of Low-Carbon Technologies*, vol. 18, pp. 896–901, Aug. 2023, ISSN: 1748-1317. DOI: 10.1093/ijlct/ctad076. eprint: https://academic.oup.com/ijlct/article-pdf/doi/10.1093/ijlct/ctad076/51176555/ctad076.pdf. [Online]. Available: https://doi.org/10.1093/ijlct/ctad076.

[3] IEA, "Global ev outlook 2023," 2022. [Online]. Available: https://www.iea.org/reports/global-ev-outlook-2023.

[4] J. Deng, C. Bae, A. Denlinger, and T. Miller, "Electric vehicles batteries: Requirements and challenges," *Joule*, vol. 4, no. 3, pp. 511–515, 2020, ISSN: 2542-4351. DOI: https://doi.org/10.1016/j.joule.2020.01.013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S254243512030043X.

[5] X. Feng, M. Ouyang, X. Liu, L. Lu, Y. Xia, and X. He, "Thermal runaway mechanism of lithium ion battery for electric vehicles: A review," *Energy Storage Materials*, vol. 10, pp. 246–267, 2018, ISSN: 2405-8297. DOI: https://doi.org/10.1016/j.ensm.2017.05.013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2405829716303464.

[6] F. Altaf, B. Egardt, and L. Johannesson Mårdh, "Load management of modular battery using model predictive control: Thermal and state-of-charge balancing," *IEEE Transactions on Control Systems Technology*, vol. 25, no. 1, pp. 47–62, 2017. DOI: 10.1109/TCST.2016.2547980.

[7] M. Torchio, "Model predictive control strategies for advanced battery management systems," 2017. [Online]. Available: https://iris.unipv.it/retrieve/e1f104fb-9c12-8c6e-e053-1005fe0aa0dd/main.pdf.

[8] G. Plett, *Battery Management Systems: Equivalent-circuit Methods* (Artech House power engineering series v. 2). Artech House, 2016, ISBN: 9781630810238.

[9] K. Harwardt, J.-H. Jung, H. Beiranvand, D. Nowotka, and M. Liserre, "Lithium-ion battery management system with reinforcement learning for balancing state of charge and cell temperature," in *2023 IEEE Belgrade PowerTech*, 2023, pp. 1–6. DOI: 10.1109/PowerTech55446.2023.10202845.

[10] Y. Yang, J. He, C. Chen, and J. Wei, "Balancing awareness fast charging control for lithium-ion battery pack using deep reinforcement learning," *IEEE Transactions on Industrial Electronics*, vol. 71, no. 4, pp. 3718–3727, 2024. DOI: 10.1109/TIE.2023.3274853.

[11] T. Duraisamy and D. Kaliyaperumal, "Machine learning-based optimal cell balancing mechanism for electric vehicle battery management system," *IEEE Access*, vol. 9, pp. 132 846–132 861, 2021. DOI: 10.1109/ACCESS.2021.3115255.

[12] Z. Xia and J. A. Abu Qahouq, "State-of-charge balancing of lithium-ion batteries with state-of-health awareness capability," *IEEE Transactions on Industry Applications*, vol. 57, no. 1, pp. 673–684, 2021. DOI: 10.1109/TIA.2020.3029755.

[13] B. Jiang, J. Tang, Y. Liu, and L. Boscaglia, "Active balancing of reconfigurable batteries using reinforcement learning algorithms," in *2023 IEEE Transportation Electrification Conference Expo (ITEC)*, 2023, pp. 1–6. DOI: 10.1109/ITEC55900.2023.10187076.

[14] G. Plett, *Battery Management Systems: Battery modeling* (Artech House power engineering and power electronics v. 1). Artech House, 2015, ISBN: 9781630810238.

[15] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, *Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor*, 2018. arXiv: 1801.01290 [cs.LG].

[16] S. Fujimoto, H. van Hoof, and D. Meger, *Addressing function approximation error in actor-critic methods*, 2018. arXiv: 1802.09477 [cs.AI].

[17] T. P. Lillicrap, J. J. Hunt, A. Pritzel, *et al.*, *Continuous control with deep reinforcement learning*, 2019. arXiv: 1509.02971 [cs.LG].

[18] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, *Proximal policy optimization algorithms*, 2017. arXiv: 1707.06347 [cs.LG].

[19] H. Choi, H. Son, Y. H. Choi, B. D. Youn, and G. Lee, "Reliability-based design optimization of a pouch battery module using gaussian process modeling in the presence of cell swelling," *Struct. Multidiscip. Optim.*, vol. 66, no. 10, Oct. 2023, ISSN: 1615-147X. DOI: 10.1007/s00158-023-03662-1. [Online]. Available: https://doi.org/10.1007/s00158-023-03662-1.

[20] R. Gauthier, A. Luscombe, T. Bond, *et al.*, "How do depth of discharge, c-rate and calendar age affect capacity retention, impedance growth, the electrodes, and the electrolyte in li-ion cells?" *Journal of The Electrochemical Society*, vol. 169, no. 2, p. 020 518, Feb. 2022. DOI: 10.1149/1945-7111/ac4b82. [Online]. Available: https://dx.doi.org/10.1149/1945-7111/ac4b82.

[21] J. Meng, M. Ricco, G. Luo, *et al.*, "An overview and comparison of online implementable soc estimation methods for lithium-ion battery," *IEEE Transactions on Industry Applications*, vol. 54, no. 2, pp. 1583–1591, 2018. DOI: 10.1109/TIA.2017.2775179.

[22] K. S. Ng, C.-S. Moo, Y.-P. Chen, and Y.-C. Hsieh, "Enhanced coulomb counting method for estimating state-of-charge and state-of-health of lithium-ion batteries," *Applied Energy*, vol. 86, no. 9, pp. 1506–1511, 2009, ISSN: 0306-2619. DOI: https://doi.org/10.1016/j.apenergy.2008.11.021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0306261908003061.

[23] D. N. How, M. Hannan, M. H. Lipu, and P. J. Ker, "State of charge estimation for lithium-ion batteries using model-based and data-driven methods: A review," *Ieee Access*, vol. 7, pp. 136 116–136 136, 2019.

[24] X. Zhu, Q. Lin, S. You, S. Chen, and Y. Hong, "A review of battery state of health estimation," in *2019 4th International Conference on Intelligent Green Building and Smart Grid (IGBSG)*, 2019, pp. 456–460. DOI: 10.1109/IGBSG.2019.8886281.

[25] X. Pichon, J. C. Crébier, D. Riu, and A. Collet, "Balancing control based on states of charge and states of health estimates at cell level," in *2015 International Conference on Clean Electrical Power (ICCEP)*, 2015, pp. 204–211. DOI: 10.1109/ICCEP.2015.7177624.

[26] A. Nath and B. Rajpathak, "Analysis of cell balancing techniques in bms for electric vehicle," in *2022 International Conference on Intelligent Controller and Computing for Smart Power (ICICCSP)*, 2022, pp. 1–6. DOI: 10.1109/ICICCSP53532.2022.9862513.

[27] M. Uzair, G. Abbas, and S. Hosain, "Characteristics of battery management systems of electric vehicles with consideration of the active and passive cell balancing process," *World Electric Vehicle Journal*, vol. 12, p. 120, Aug. 2021. DOI: 10.3390/wevj12030120.

[28] R. R. Kumar, C. Bharatiraja, K. Udhayakumar, S. Devakirubakaran, K. S. Sekar, and L. Mihet-Popa, "Advances in batteries, battery modeling, battery management system, battery thermal management, soc, soh, and charge/discharge characteristics in ev applications," *IEEE Access*, vol. 11, pp. 105 761–105 809, 2023. DOI: 10.1109/ACCESS.2023.3318121.

[29] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[30] S. Ishii, W. Yoshida, and J. Yoshimoto, "Control of exploitation–exploration meta-parameter in reinforcement learning," *Neural Networks*, vol. 15, no. 4, pp. 665–687, 2002, ISSN: 0893-6080. DOI: https://doi.org/10.1016/S0893-6080(02)00056-4. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0893608002000564.

[31] K. Ochi and M. Kamiura, "Overtaking method based on variance of values: Resolving the exploration–exploitation dilemma," *Procedia Computer Science*, vol. 24, pp. 126–136, 2013, 17th Asia Pacific Symposium on Intelligent and Evolutionary Systems, IES2013, ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2013.10.035. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050913011770.

[32] T. M. Moerland, J. Broekens, A. Plaat, and C. M. Jonker, *Model-based reinforcement learning: A survey*, 2022. arXiv: 2006.16712 [cs.LG].

[33] V. Mnih, K. Kavukcuoglu, D. Silver, *et al.*, *Playing atari with deep reinforcement learning*, 2013. arXiv: 1312.5602 [cs.LG].

[34] M. Ahmed, *Electric vehicle dynamics simulation - ev_sim*, 2023. [Online]. Available: https://github.com/m0in92/EV_sim.

[35] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-baselines3: Reliable reinforcement learning implementations," *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021. [Online]. Available: http://jmlr.org/papers/v22/20-1364.html.

[36] M. Towers, J. K. Terry, A. Kwiatkowski, *et al.*, *Gymnasium*, Mar. 2023. DOI: 10.5281/zenodo.8127026. [Online]. Available: https://zenodo.org/record/8127025 (visited on 07/08/2023).

[37] Martín Abadi, Ashish Agarwal, Paul Barham, *et al.*, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: https://www.tensorflow.org/.

[38] MathWorks. "Overview of memory-mapping - matlab simulink - mathworks nordic." (2024), [Online]. Available: https://se.mathworks.com/help/matlab/import_export/overview-of-memory-mapping.html.

[39] *Thesis work source code - rl_balancing*, 2024. [Online]. Available: https://github.com/giomazzolo/rl_balancing.

A

Appendix

A.1 Python Cell Simulation Execution Example

Below is a Python example of how to set up, initialize and run a cell pack simulation. The example first configures the simulation parameters and also chooses the cell model, then it starts the simulation. Then it runs the simulation step by step and saves SOC, SOH and simulation status on an array. Once the simulation finishes, we reset it and start a new simulation run. Lastly, the simulation is stopped. Source code is available on the projects github repository [39].

```
# Imports
import sys
import numpy as np
sys.path.insert(1, "../../cell_simulation")
import SimCellPack as scp

# Set cell and simulation parameters
cellModel="P14"
balancing = "active"
utilization = [10,2]
simCycles = 10
seed = 1
profile = "us06"
sampleFactor = 30
numCells = 10
getSOCsWhen = "all"

# Create cell simulation class and pass
# the parameters
cellModel = scp.SimCellPack(
    cellModel=cellModel,
    numCells = numCells,
    simCycles = simCycles,
    seed = seed,
    profile = profile,
```

```
26          balancing = balancing,
27          getSOCsWhen = getSOCsWhen,
28          sampleFactor = sampleFactor,
29          utilization = utilization)
30
31      # Start simulation
32      cellModel.startSim()
33
34      # Balancing feedback - defined by balancing formula.
35      # If simualting "active" balancing, feedback
36      # must be all 1s for no balancing
37      # If simualting "passive" balancing, feedback
38      # must be all 0s for no balancing
39      feedback_nop = [int(balancing=="active")]*numCells
40
41      # Array to save all the simulation states for future use
42      simStatesArr = []
43
44      # Array to save all the SOCs
45      # Each index contains an array of SOCs
46      data_soc = []
47
48      # Array to save all the SOHs
49      #Each index contains an array of SOHs
50      data_soh = []
51
52      # Initialize code to start the loop
53      code = 1
54      # Code coding:
55      #   0 - simulation waiting
56      #   1 - discharging
57      #   2 - charging
58      #   3 - resting
59      #   >= 4 - unassingned
60
61      # While simulation not waiting, get simulation step
62      while code > 0 and code < 4:
63
64          # Get a simulation state, soc and soh
65          code, state = cellModel.getSimStep()
66
67          # Simulation cannot continue until feedback is sent
68          # Feedback must be sent at each step
69          # to continue the simulation
70          # Feedback is ignored when the simulation is not
71          # in its respective balancing phase
```

```
72
73      # Save simulation state
74      simStatesArr.append(code)
75
76      # Save SOC in array, for later analysis
77      data_soc.append(state[0:cellModel.numCells])
78
79      # Save SOH in analysis
80      data_soh.append(state[cellModel.numCells:])
81
82      # If simulation not waiting
83      if code > 0 and code < 4:
84          cellModel.sendSimFeedback(feedback_nop)
85
86  # Resets the simulation and start over
87  cellModel.resetSim()
88
89  # We let the simulation run and finish, but we save nothing
90  code = 1
91  while code > 0:
92       code, state = cellModel.getSimStep()
93       if code > 0:
94           cellModel.sendSimFeedback(feedback_nop)
95
96  # Stops the simulation, kills processes and cleans files
97  cellModel.stopSim()
98  del cellModel
```

A.2 Additional Simulation Runs

Figure A.1: Active SOC balancing test - 1000 cycles - utilization [10, 2].

Figure A.2: Active SOC balancing test - 1000 cycles - utilization [20, 2].