Battery Monitoring System & State of Charge Estimation Modelling

Name: Daniel Woodhall
Date: 10/06/2020

Abstract

This report presents research that was undertaken on lithium-ion batteries and outlines several different methods to determine the state of charge (SoC) of a lithium-ion battery. The methods include traditional methods such as direct comparison between the open-circuit voltage (OCV) and SoC and coulomb counting. In this report improved methods to estimate the state of charge are proposed, namely machine learning models that can be trained and used to estimate the state of charge. Four such models were made: a Kalman filter, a Random Forest Regression model, a Support Vector Regression model, and a Neural Network. Of the four models proposed, the Neural Network showed the best performance, giving a mean absolute error of 6.96%. Due to time limitations and the computational power available, not all of the models have been tuned to their maximum potential, however the process for how this would be done has been outlined. The Neural Network produced is a robust model that would be able to be used under varying temperature conditions. The models were all trained on a dataset provided by Dr. Phillip J. Kollmeyer of McMaster University. A battery monitoring system was also designed to allow for a dataset to be constructed that could be used to train these models on the battery that will be used for the Mars Rover challenge 2020.

Declaration and Acknowledgments

I declare that the work undertaken and outlined in this report is entirely my own work and where information has been taken from other sources it has been appropriately referenced.

Acknowledgments and thanks go to Dr. Phillip J. Kollmeyer of McMaster University for allowing the use of his Panasonic 18650PF dataset that was used to train all of the models shown in this report.

Table of Contents

1. Introduction

State of charge estimation in batteries is an extremely important yet seemingly poorly understood process. It is vital to be able to estimate the state of charge of a battery to ensure reliable operation, especially in electrical vehicles such as a rover used for space missions. State of charge can be found from the open circuit voltage of a battery, however this has severe limitations associated with it. In recent years, great advancements have been made towards improving the machine learning models that are now used to estimate the state of charge; namely the use of neural networks. Previous estimation models incorporated Kalman filters that were used either on their own or in combination with traditional regression models, but the use of neural networks to estimate state of charge shows improved results when compared to these more traditional methods. The state of charge shows the relative remaining capacity of a battery which can in turn be used to estimate the remaining run-time of the battery.

The older models used to estimate state of charge are limited in that they often cannot account for variations in temperature without vast amounts of additional data. This report looks into the use of various state of charge estimation methods and their limitations. These methods are compared and an attempt to find the most suitable model for this estimation is made.

Due to time constraints and the computational power available not all models were able to be fully finished and optimised, however the code for these optimisations has been included and is shown in Appendix A.

2. Project Aims

- Undertake research to gain a strong understanding of lithium-ion batteries, how they work and their limitations.

- Undertake research to gain an understanding of current state of charge estimation models as well as how they work and the mathematics that make them work.

- Design a battery monitoring system that could be used to make a dataset to train various state of charge estimation models as well as give live measurements to the constructed models for use in the Mars Rover challenge. The battery monitoring system should be designed to be as cheap as possible, whilst still giving accurate results. A bill of materials must be made for the battery monitoring system.

- Construct various state of charge estimation models using more traditional techniques such as Kalman filters along with more revolutionary techniques such as neural networks.

- Draw comparisons between the constructed models to determine which would be the most appropriate for use in the Mars Rover Challenge.

3. Literature Review

3.1 Lithium-ion Batteries

3.1.1 Lithium-ion Battery Chemistry

Modern batteries create power using three main components, a cathode with a positive charge that attracts electrons, an anode with a negative charge that releases stored electrons, and an electrolyte that facilitates the movement of electrons between the anode and cathode; it's this flow of electrons that creates a charge [1].

Lithium batteries were first pioneered in 1912 by G.N Lewis, although they didn't become commercially available until the 1970's due to the especially unstable nature of Lithium as a metal [2]. This instability in the original lithium battery designs that used metal-lithium anodes caused them to be rejected in favour of a graphite anode and a lithium metal oxide cathode, largely due to the need for rechargeable batteries; attempting to recharge the original metal-lithium anode batteries caused excessive dendrites to form on the anode which greatly reduced the performance and capacity of the batteries [3] [4].

Modern lithium-ion batteries, as previously stated, utilise a graphite anode and lithium metal oxide cathode with an electrolyte comprised of a lithium salt, usually $LiPF_6$, in an organic solution with other additives to improve the stability of the solution [5]. The graphite anode has a molecular structure that allows lithium ions to intercalate between the layers of graphite atoms and bind, with each lithium ion requiring six graphite atoms to bind to [6] [7]. In recent years silicon has been used, dispersed in between the graphite layers, to improve the efficiency of the batteries. The silicon improves the efficiency as it has a higher specific capacity than graphite and only requires four silicon atoms to bind to each lithium ion, allowing for a higher density of lithium ions within the molecular structure [8] [9].

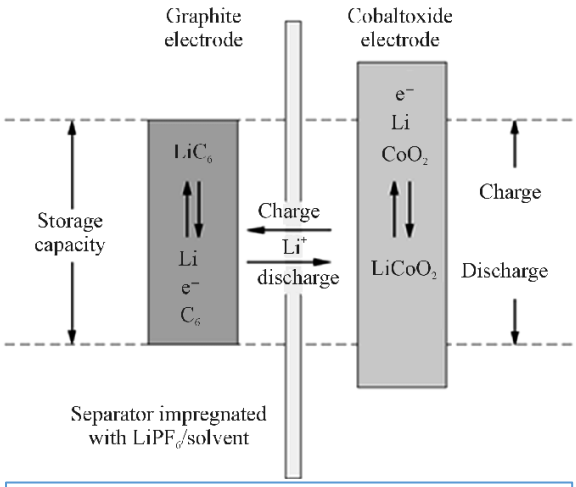


Figure 1 - Diagram of Lithium Cobalt Oxide battery (Peter H L Notten, 2020)

There are many different types of lithium-ion batteries, differing by the cathodic material, this report will focus on the most common type of lithium-ion battery – lithium cobalt oxide ($LiCoO_2$). The structure of a $LiCoO_2$ battery is shown in Figure 1. $LiCoO_2$ batteries have a nominal voltage of 3.60V. The chemical reaction at the anode for a $LiCoO_2$ battery during discharge, where lithium is oxidised, is shown in equation (1).

$$LiC_6 \longrightarrow xLi^+ + xe^- + C_6 \quad (1)$$

During discharge, lithium dioxide at the cathode absorbs lithium ions, in turn gaining electrons from the electrolyte as shown in equation (2).

$$Li_{1-x}CoO_2 + xLi^+ + xe^- \longrightarrow LiCoO_2 \quad (2)$$

Combining equation (1) and equation (2) gives the overall chemical equation shown in equation (3) [10].

$$C_6 + LiCoO_2 \longleftrightarrow Li_xC_6 + Li_{1-x}CoO_2 \quad (3)$$

In equation (3), the forward reaction represents charging and the backwards reaction represents discharging.

3.1.2 Limitations of Li-ion Batteries

Although lithium-ion batteries give distinct advantages over older battery chemistries such as higher energy density, higher power density, and longer service life, there are some severe limitations and disadvantages associated with them. The main issue with li-ion batteries is the safety risk, if they are exposed to high temperatures or severely over charged they are prone to thermal runaway which can lead to fire and/or explosion [11]. When a li-ion cell explodes or combusts it releases hydrogen fluoride (HF) and phosphoryl fluoride (POF_3) gasses, both of which are extremely toxic and can be absorbed through the skin as opposed to needing to be directly inhaled [12]. Due to the severity of a li-ion battery malfunctioning, safety mechanisms must be put in place. These safety systems can be something as simple as a one-shot fuse that completely disconnects the battery if it's becoming too hot or is exposed to too high a current or voltage, to entire battery monitoring systems that carefully monitor the temperature, incoming and outgoing current, and voltage; these are again able to disconnect the battery if certain values are exceeded through the use of a protection circuit [13] [14].

Another limitation of li-ion batteries is their performance when subject to abnormal environmental temperatures. As already mentioned high temperatures can adversely affect the batteries causing thermal runaway. In addition, high temperatures have also been shown to decrease the cycle life of the battery [15]. Low temperatures also negatively affect li-ion batteries, although not with the same catastrophic consequences as high temperatures. Li-ion batteries exposed to low temperatures (below -10°C) have greatly increased internal resistance that can cause a noticeable drop in output voltage, especially if a low-resistance load is attached [16]. In addition to this, charging a li-ion battery at these low temperatures can cause severely shortened cycle life and in some cases complete cell failure [17] [18].

The state of health (SoH) and state of charge (SoC) of a battery are always important parameters regardless of the battery type. The SoH gives information on the relative capacity of the battery compared to its original state; for example if a battery has 80% SoH and a nominal capacity of 2Ah, the actual maximum capacity of the battery would be 1.6Ah. The SoC gives an indication of the remaining capacity of the battery compared to its current maximum capacity. SoC can be determined through several methods, the easiest of these is measuring the open circuit voltage (OCV) and comparing it to the discharge profile for the battery [19]; an example discharge profile, discharging at 1C is shown in Figure 2. As can be seen in Figure 2, the discharge profile for a li-ion battery appears to be linear, with quadratic sections at the beginning and end of the profile. When the battery is at rest determining the SoC from the OCV is a fairly accurate method. However, once the battery is subject to a load or is charging the OCV experiences fluctuations which can make determining an accurate value for the SoC more difficult. The OCV also varies significantly when the battery is subject to abnormal external temperatures. As such, this method is considered to be the least reliable and most sophisticated systems use an alternative method to determine the SoC.

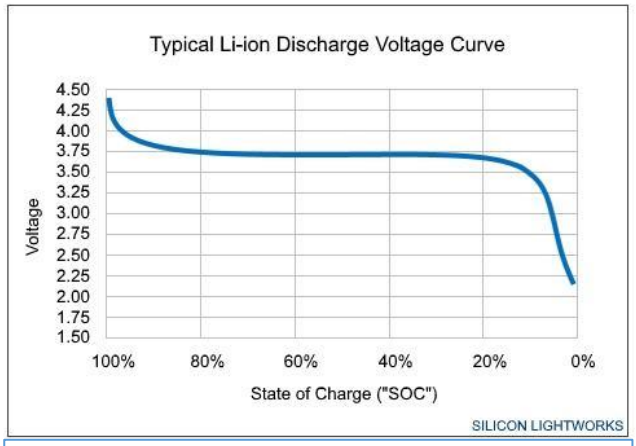


Figure 2 - Typical Li-ion Discharge curve (Silicon Lightworks, 2016)

A more accurate method of measuring the SoC at any given time is coulomb counting. A coulomb counter is a device that continually monitors the current, totalling it over time and sending a pulse to a microcontroller or computer once a set amount of ampere-hours (Ah) have been used. The amount of Ah that have been used over time can then be subtracted from the total Ah rating of the battery, giving the remaining capacity [20]. There are several additional factors that need to be considered when finding the SoC from coulomb counting such as the coulombic efficiency of the battery; fortunately the coulombic efficiency of li-ion batteries is high, usually around 99%. The overall equation for finding the SoC from coulomb counting is shown in equation (4), where SoC(t) is the SoC being estimated, $SoC(t_0)$ is the previous known SoC measurement, η is the coulombic efficiency of the battery, I is the current, C^o_b is the reading from the coulomb counter, and dT is the time period.

$$SoC(t) = SoC(t_0) - \int_{t_0}^{t} \frac{\eta(T)I(T)}{C_b^0} dT \quad (4)$$

As with the OCV method, there are limitations associated with coulomb counting. If the current reading, *I*, is off by only a small amount it can result in large errors when integrated over a significant time period. In addition to this, if the battery is not at a known SoC level when the measurements started, there is no way to determine the current SoC; the battery is only ever at a known SoC value when it is fully charged or discharged. To counter this, the OCV can be used in conjunction with the measurements from coulomb counting to give a more accurate estimate. The OCV measurement is still subject to the limitations previously discussed and as such the battery needs to be rested for several hours to get an accurate reading; using an unsettled OCV measurement will give large errors in the SoC. This combined method is a definite improvement over using either one individually but is still not robust enough to give accurate estimations if the required conditions are not met [21]. Improved methods for SoC estimation are discussed in **3.2**.

3.2 SoC Estimation Methods

3.2.1 Kalman Filters

Kalman filters (KF) are a widely used iterative estimation method that allow for an unknown value to be estimated from measured values that contain a large amount of statistical noise. This is an especially useful method when trying to gain estimates from electrical sensors that commonly contain a large amount of noise in their outputs. In the context of SoC estimation a KF can be used to give a much more accurate estimate of the OCV whilst the battery is under load, eliminating some of the fluctuations in the OCV that li-ion batteries experience during use; thus allowing for the estimated OCV to be used to find the relevant SoC for the battery. A KF can use multiple measurements as inputs, such as current and temperature, to give an improved estimate for the OCV using matrix manipulation [22].

A specific type of KF, known as an Extended Kalman Filter (EKF), can be used to give estimates for nonlinear data by linearising around an estimate of the mean and covariance [23]; this is especially useful for estimating the SoC from the OCV which only follows a linear profile in-between approximately 80% and 20% SoC. This method could be used in combination with the coulomb counting method described in **3.1.2** to give a better approximation of the starting SoC, even when the battery has a load applied. However there are still issues with this method, estimating the SoC from the KF estimate for OCV has inherent error as the relationship between OCV and SoC follows a specific discharge profile for each

battery type. The OCV changes very little between 80% and 20% SoC for most li-ion batteries meaning very precise measurements for the OCV are required to determine a good estimate for the SoC; this can require expensive voltage sensors and implementing this in the Mars Rover design would be especially challenging due to budget limitations [24].

A flowchart of the processes in a KF is shown in Figure 3, with all the relevant variables labelled. A KF uses an iterative process, feeding back previous estimates and the error associated with these estimates, as well as values from the sensors and the error associated with these values, to give better estimates on the next iterations. The error associated with these values allows for the Kalman Gain (K) to be calculated using equation (5), which is a simplified version of the equation for K in Figure 3.

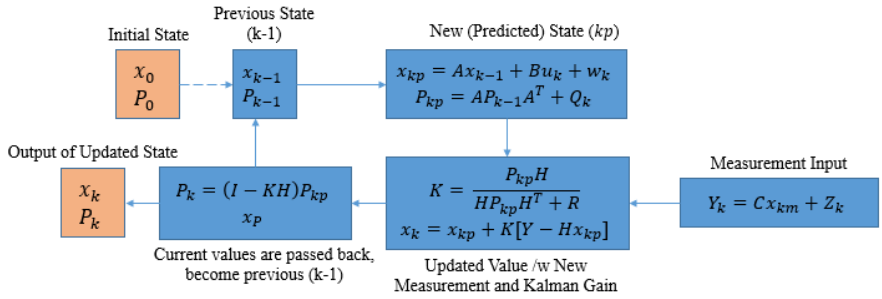


Figure 3 - Flowchart of multi-dimensional KF processes: K represents Kalman Gain, x represents State Matrix, P represents Process Covariance Matrix, u represents Control Variable Matrix, w represents Predicted State Noise Matrix, Q represents Process Noise Covariance Matrix, Z represents Measurement Noise, Y represents the Measurement from Sensors, and R represents the Sensor Noise Covariance Matrix.

$$K = \frac{E_{Est}}{E_{Est} + E_{Mea}} \quad (5)$$

Equation (5), where E_{Est} is the error from the previous estimate and E_{Mea} is the error from the previous measured value, gives a weighting based off the previous errors which is then used to improve the estimate on the next iteration. The basic equation for calculating the current estimate is shown in equation (6) where Est represents the estimate, Mea represents the measurement, t represents the current time step and t-1 represents the previous time step.

$$Est_t = Est_{t-1} + K(Mea - Est_{t-1}) \quad (6)$$

Equation (7) below shows how the error for the estimate is calculated. A simplified version of equation (7) is shown in equation (8).

$$E_{Est_t} = \frac{(E_{Mea})(E_{Est_{t-1}})}{E_{Mea} + E_{Est_{t-1}}} \implies E_{Est_t} = (1 - K)\left(E_{Est_{t-1}}\right) \quad (7), (8)$$

3.2.2 Random Forest Regression

Random forest models (RF) are supervised ensemble learning methods that use a "bagging" technique as opposed to a "boosting" technique. The random forest regression model consists of multiple decision trees that gives the mean prediction of all of the decision trees as the output. The decision trees consist of three types of nodes: the root node which is the first node in the tree and often is the most important variable in determining the output, internal nodes which represent the decision state, and leaf nodes which represent the output at the end of the decision tree. The order of nodes in the decision trees is an important factor that can affect the accuracy of the model, with variables that have the highest direct correlation to the output being placed further up the tree. Impurity in the nodes is where the variables do not have a direct or obvious correlation to the output being calculated. The impurity for each node can be calculated using the Gini Impurity equation shown as equation (9).

$$Impurity = 1 - (Probability\ of\ positive\ outcome)^2 - (Probability\ of\ negative\ outcome)^2 \quad (9)$$

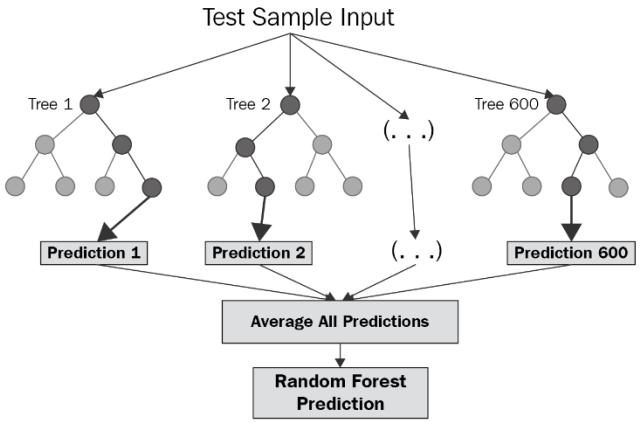


Figure 4 - Outline of a Random Forest Model (Afroz Chakure, 2019)

From this equation the total impurity for that decision tree can be calculated from the weighted average of the Gini Impurity for each individual output of the decision tree; with a lower impurity value suggesting that the tree will give a better estimate [25]. The decision trees can then be weighted in terms of accuracy based on their Gini impurity value to give a better average estimate.

When a RF is set up, the dataset that will be used to train it must be "bagged" (bootstrap aggregating) into what is known as a bootstrapped dataset and an out-of-bag-dataset. The bootstrapped dataset is constructed by selecting random rows from the original dataset (duplicate rows are allowed) and arranging them in a random order, this bootstrapped dataset is then used to train the model. Approximately 1/3 of the data from the original dataset will not be used in this bootstrapped dataset, and these unused entries become the out-of-bag-dataset. Once the model has been trained, the out-of-bag-dataset is used to validate the model by using data that is completely new, which then allows for the error of the model to be calculated by comparing the estimates to the known outputs [26] [27]. The RF model can be optimised by changing the amount of variables that each tree contains as well as which variables are used, assessing the error for each to find the optimal combination of variables. The number of estimators (decision trees) can also be varied to assess how the error changes; generally the error will be reduced the more decision trees there are in the model, however this also requires more computational power. A diagram showing the rough structure of a RF model is shown in Figure 4.

In the case of SoC estimation several variables would be fed into the model from a battery monitoring system, the most important of these variables being the OCV. Other important variables to monitor would be the temperature of the battery and the external temperature as these both affect the OCV and performance of the battery as discussed in **3.1.2**. To improve the model further, a Kalman Filter can be used as described in **3.2.1**. The KF could be used to give a more accurate approximation of the OCV with this approximation then being used by the RF model to estimate the SoC of the battery, thus eliminating a lot of the noise from the original sensor readings before the data is inputted to the model [28].

3.2.3 Support Vector Regression

Support vector regression (SVR) is a Kernel method that can be used to solve regression problems. In this instance it would be used to find the SoC from the various sensor measurements such as OCV, current, temperature and external temperature. There are four main terms associated with SVRs: Kernels, hyper plane, boundary line, and support vectors. A Kernel method is any method that is used to map lower dimensional data into a higher dimension to allow for complex calculations to be performed. The hyper plane refers to the line or plane that allows for predictions of the target value to be made by attempting to fit the predictions to the plane. Boundary lines create a margin either side of the hyper plane, mirroring the profile of the hyper plane, that allow for misclassifications or outliers in the data to be identified. A support vector refers to the data points that are closest to the boundary lines, with the distance being the minimum amount [29].

SVR differs from traditional simple regression as it doesn't try to minimise the error rate, instead it attempts to fix the error within a threshold value. The boundary lines either side of the hyper plane can be referred to as +ε and –ε respectively. The equation of the hyper plane is shown to be $wx + b = 0$, with the boundary line equations being $wx + b = +\varepsilon$ and $wx + b = -\varepsilon$ for the top and bottom boundary lines respectively. This gives the total equation for a linear hyper plane as shown in equation (10).

$$-\varepsilon \leq y - wx - b \leq +\varepsilon \quad (10)$$

The most commonly used kernel function is known as a polynomial kernel, however for SoC estimation the radial basis function (RBF) is more commonly used as it is easier to calibrate and tune and allows for nonlinear relationships to be modelled with more accuracy [30]. The RBF works by performing calculations on the data in infinite dimensions and behaves much like a weighted nearest neighbour model, where the closest observations have the biggest influence on the output. Use of a kernel reduces computational power needed as it doesn't actually convert the points into a higher dimension, it only calculates the relationships between points as if they were in a higher dimension. The RBF calculation is shown in equation (11), where a and b refer to two different measurements and γ is a scalar for the relative influence.

$$e^{-\gamma(a-b)^2} \quad (11)$$

When the points a and b are very far apart the resulting influence value is very close to 0; with a lower influence value meaning the values have less effect on each other. An example of how the RBF kernel works will be shown below. Starting with equation (11), setting gamma as ½ and computing the output. This gives equation (12) shown below.

$$e^{-\frac{1}{2}(a-b)^2} = e^{-\frac{1}{2}(a^2+b^2)} e^{ab} \quad (12)$$

Taking the Taylor series expansion of the last term, e^{ab}, gives equation (13) below.

$$e^{ab} = 1 + \frac{1}{1!}ab + \frac{1}{2!}(ab)^2 + \frac{1}{3!}(ab)^3 + \cdots + \frac{1}{\infty!}(ab)^{\infty} \quad (13)$$

It can be shown that a polynomial kernel of $r = 0$ and $d = 0$ is equal to 1, with $d = 1$ it is equal to the second term in equation (13) and so on for all values of d. Thus, it can be shown that each term in the Taylor series expansion for the last term of the RBF contains a polynomial kernel with $r = 0$ and values of d increasing from 0 to infinity. This gives the dot product of e^{ab} as shown in equation (14).

$$e^{ab} = \left(1, \sqrt{1/1!}\,a, \sqrt{1/2!}\,a^2, \ldots, \sqrt{1/\infty!}\,a^{\infty}\right) \cdot \left(1, \sqrt{1/1!}\,b, \sqrt{1/2!}\,b^2, \ldots, \sqrt{1/\infty!}\,b^{\infty}\right) \quad (14)$$

Going back to the original RBF equation, the dot product for e^{ab} can be substituted in place of the original e^{ab}. The square root for the first term can be substituted in as well, for simplicity this will be labelled as S. This gives the dot product shown in equation (15).

$$e^{-\frac{1}{2}(a-b)^2} = \left(s, s\sqrt{1/1!}\,a, s\sqrt{1/2!}\,a^2, \ldots, s\sqrt{1/\infty!}\,a^{\infty}\right) \cdot \left(s, s\sqrt{1/1!}\,b, s\sqrt{1/2!}\,b^2, \ldots, s\sqrt{1/\infty!}\,b^{\infty}\right) \quad (15)$$

This means that when values are plugged into the RBF, the value at the end is the relationship between the two points in infinite dimensions [31].

The SVR model can be improved by adjusting the value of ε for the boundary lines to include a greater or fewer amount of data points. SVR can be thought of as if each data entry in the training data represents its own dimension, when the RBF is evaluated between a test point and a point in the training dataset, the resulting value gives the coordinate of the test point in that relative dimension. The vector given when the test point is evaluated against all points in the training set is the representation of the test point in the higher dimensional space. Thus, once this vector is found it can be used to perform regression to calculate the output, giving an estimate of the desired value [32].

3.2.4 Neural Networks

Neural networks (NN), often referred to as deep-learning, are an attempt at replicating neurons inside a human brain; allowing the network to learn and make decisions. A NN is usually comprised of 3 different types of layers: an input layer, an output layer, and one or several hidden layers. As the names suggest, the input layer contains the input variables to the problem and the output layer is where the estimated value is outputted. The hidden layers contain interconnected processing nodes where data is transferred from the layer of nodes below (with the input layer being the bottom layer) to the layer of nodes above. Each node is assigned a weight value and when data is received it multiplies it by the associated weight value. These results are then summed to give a single output, if this value is above a set threshold value the neuron activates and sends its value further up the network along all of its connections [33].

When a NN is initially being trained the weights and thresholds for each node are randomly generated. The data from the training dataset is fed from the input layer upwards through the successive layers, being multiplied and summed as it goes, until it reaches the output layer and a final estimate is given. During the training process, the weights and thresholds for the nodes are automatically adjusted until the error between the estimated output and desired value is minimised. As with the previous models discussed, the inputs for the neural network in this instance would be the OCV, current, temperature, and ambient temperature measurements.

The basic structure of a NN is shown in Figure 5, displaying the 3 different types of layers. NNs can be improved for each specific task by changing the number of hidden layers and nodes per layer in the model. For simple problems generally 1-5 hidden layers are sufficient. Adding more hidden layers does not necessarily improve the accuracy of the model as it can cause the NN to over-fit the result. In the same manner, too few hidden layers can result in an under-fit estimate, thus the optimal number of hidden layers must be found for each individual problem. The same applies for the amount of nodes per hidden layer. There is no clear way to determine the optimal number of hidden layers or nodes per layer in the model and as such this is usually found by trial and error. There are a few basic rules that can be used as a starting point to work off, however many of these rules directly contradict each other as NNs are so poorly understood [34].

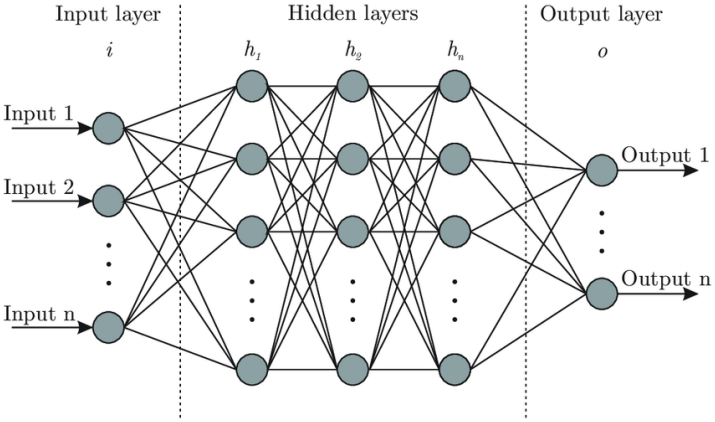


Figure 5 - Basic Neural Network Structure (Lavanya Shukla, 2019)

One rule states that the number of nodes per layer should start off somewhere in between the number of inputs and the desired number of outputs. In the instance of this SoC problem, that would put the optimal number of nodes per layer somewhere in between 1 and 4. Another rule uses a basic equation to calculate a maximum number of nodes before the model is over-fit, this is shown in equation (16), where N_h is the maximum number of nodes, N_i is the number of input nodes, N_o is the number of output nodes, N_s is the number of samples in the data set used to train the model, and α is an arbitrary scalar that is usually set to be between 2-10.

$$N_h = \frac{N_s}{\alpha(N_i+N_o)} \quad (16)$$

This rule would give a vastly different number of neurons than the first rule stated as the data set used to train NNs is often extremely large; commonly including more than 100,000 data points [35].

Another way to improve the accuracy of a NN is to change the amount of epochs used when training the network. One epoch equates to one cycle through the entire training dataset. Generally the error will be reduced the more epochs are used when training the model up until the point where the model becomes over-fit. Once the model becomes over-fit, the error will increase. Increasing the amount of epochs also vastly increases the amount of computational power needed, with 2 epochs taking twice the amount of time to train as 1 epoch. As such, a balance between computing time and minimising the error must be found [36].

4. Proposed Solutions

4.1 Circuit and BMS

To allow the models created to be used for the Mars Rover challenge, a battery monitoring system (BMS) must first be designed to allow the relevant data to be fed into the models. This BMS would be used during the challenge to allow for the chosen model to predict the SoC during use, as well as before the challenge to gather data about the specific battery being used (An 18V DeWalt drill battery) to allow for the model to be trained. As discussed in **3.1.2,** the main parameters of the battery that would need to be monitored would be the OCV, the current draw, the temperature of the battery, and the ambient temperature. A protection circuit for the battery will not be required as the battery being used already includes one.

The measurements would be done using several sensors connected to an Arduino Uno. A coulomb counter would be used in conjunction with an ammeter to give an accurate measurement of the total Ah draw over time as well as the current being drawn at each moment. Two temperature sensors would be used, one mounted directly onto the casing of the battery to monitor the battery temperature and one mounted further away from the battery to monitor the ambient temperature. A voltage divider would also be used, with the wires connected directly onto the positive and negative terminals of the battery to monitor the OCV without any interference from other components. A connection would then be run from the voltage divider to one of the analog inputs of the Arduino. A voltage divider is needed for the Arduino to read the OCV as it has a maximum reading of 5V, thus the voltage divider is used to scale down the voltage from 18V to a value that is readable by the Arduino. The voltage divider also includes a 100nF capacitor that acts as a low-pass filter to remove some of the noise from the readings. The voltage divider scales the voltage down as shown in equation (17) [37].

$$V_{Out} = V_{In}\, R2/(R1 + R2) \Longrightarrow V_{Out} = V_{In}\, 10000/36000$$

This will scale the voltage down by multiplying it by 5/18, meaning that a voltage reading of 18V (The batteries maximum voltage) will be the equivalent to a reading of 5V on the Arduino. Alternatively, a more sophisticated voltage sensor could be used to feed its results directly into the Arduino; however a voltage divider was used due to trying to reduce the total cost. The circuit diagram for the BMS is shown in Figure 6, with red lines representing power lines, black lines representing ground wires and green lines representing data lines.

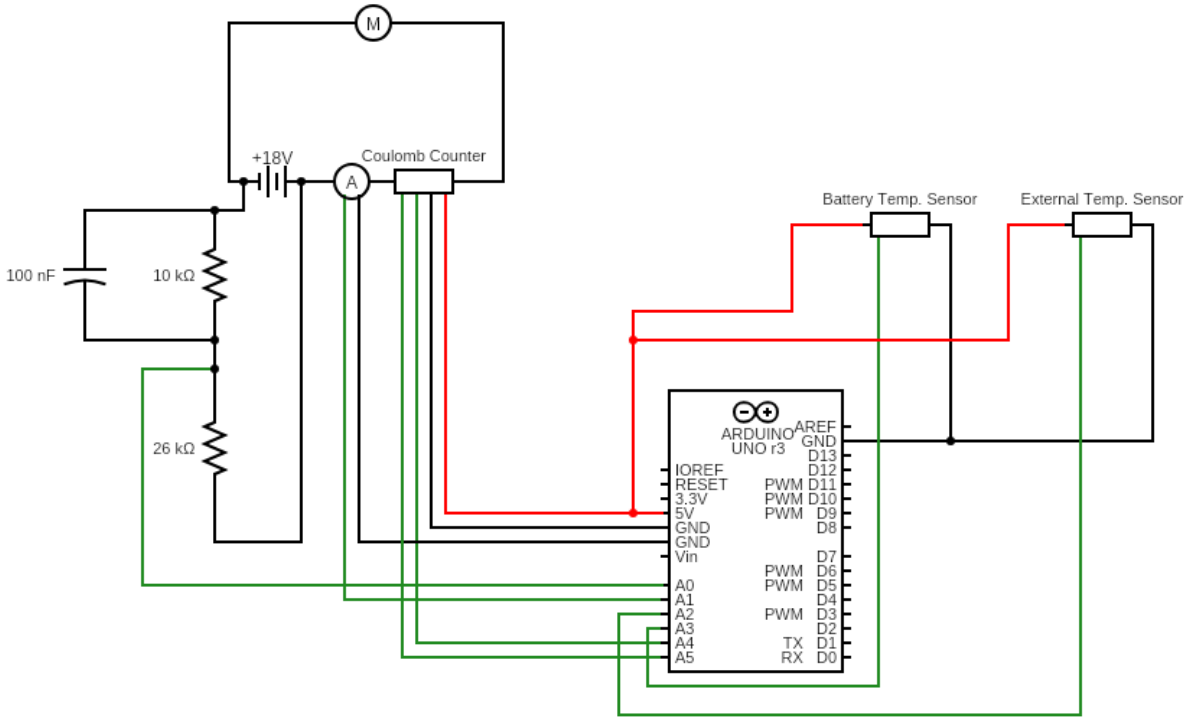


Figure 6 - BMS Circuit Diagram

This circuit could be used to produce a dataset for training the models by data-logging each measurement over time whilst the battery is subject to a load. The battery would need to start at 100% SoC during these measurements and the coulomb counter reading would be used to determine the SoC by subtracting the total Ah draw over time from the known capacity of the battery. A more accurate capacity for the battery could be found by fully charging the battery then fully discharging it at a known current whilst monitoring the time for a full discharge. The capacity in Ah is then equal to the current multiplied by the time it took to fully discharge the battery in hours. The data would be logged by connecting the Arduino directly to a PC or wirelessly through the use of an XBee, then transferring the data measurements into a CSV file. During the actual Mars Rover challenge an XBee would be used to feed the measurements back to a PC to give live updates on the SoC during use.

4.2 Dataset

Due to the Covid-19 pandemic the BMS circuit was unable to be made and tested. As such, a dataset was unable to be produced for the DeWalt battery used. In lieu of this, a dataset, created by Dr. Phillip Kollmeyer of McMaster University, for a Panasonic 18650PF li-ion battery was used to train and validate the models that were created [38]. The models would still be able to function for the Mars Rover challenge once a dataset was created and then used to train the models as outlined in **4.1**. The models would need to be re-tuned to find the optimal parameters in the same way they have been tuned in the following sections.

The dataset used for the models included a variable drive cycle performed on a Panasonic 18650PF li-ion battery. The drive cycles performed included regenerative breaking and as such allowed the models to be trained to estimate the SoC of the battery when both charging and discharging. The dataset included 109,641 data entries which is more than sufficient to initially train the models. As Python was being used to construct the models instead of MATLAB, the dataset (which was initially in .mat format) was converted into CSV format to allow for it to be read. The dataset was also rearranged to allow for easier manipulation of the data. An additional column, Capacity, was added to the dataset which was calculated from the known capacity of the battery and the Ah measurement as described in **3.1.2**. The subsequent drive cycle data sets were then used to validate the model which showed the least error. These datasets combined included 1,126,083 data entries. As such a large amount of data entries were used there was high confidence that the outputted error was accurate.

Before any of the models were made, all of the data from the first dataset was visualised to show any obvious correlations between values that could help when deciding which variables to include in the predictions. This visualisation also aided with the tuning of the models and the decision on which variables should be given the highest weighting for the estimates. The visualised data is shown in Figure 7. One of the most interesting findings from visualising this data was the correlation between the battery temperature and OCV. As can be seen, the battery temperature increases as the OCV decreases, suggesting that the temperature would be an extremely important variable in estimating the SoC as this directly correlates to the OCV. However, this increase in temperature could also be due to the battery being in use for an extended period of time. The fluctuation in OCV measurements is also clearly shown in this data, solidifying the need for a KF.

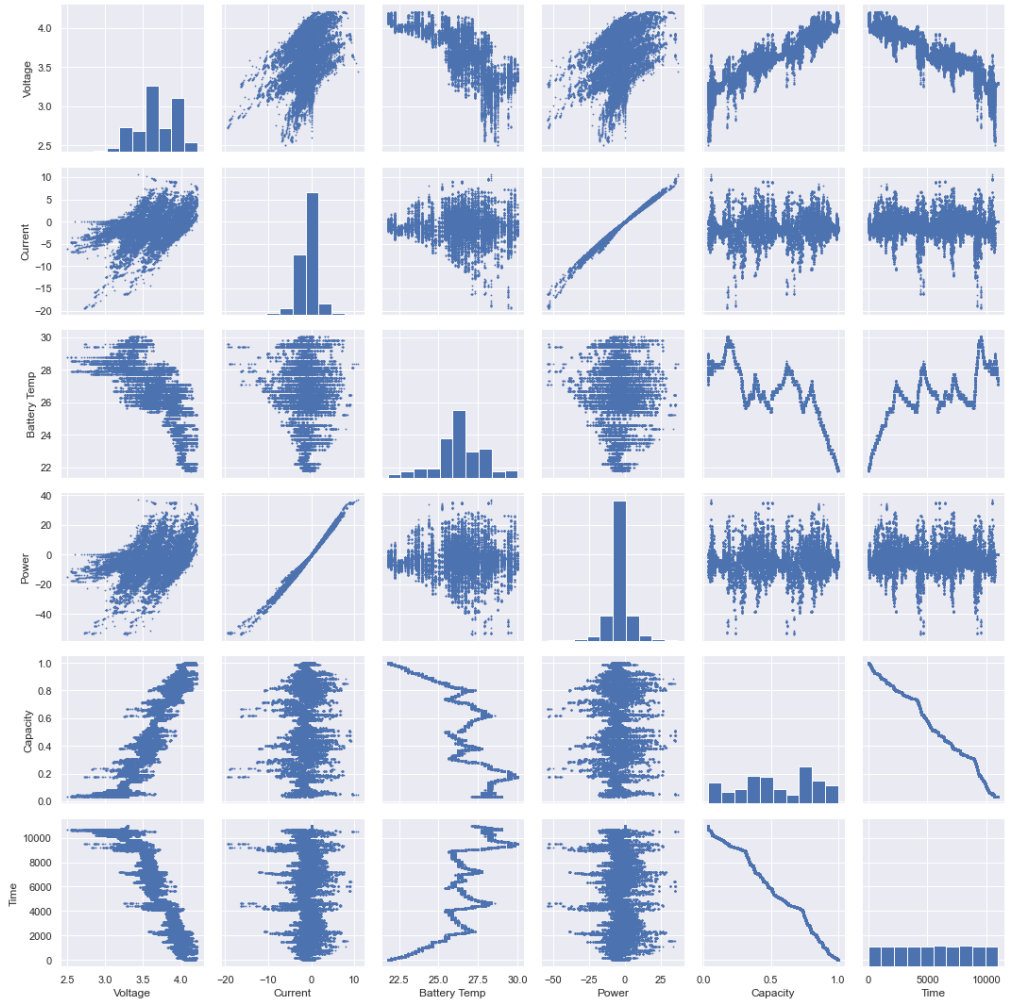


Figure 7 - Visualised Data from the 1st Dataset

The dataset was checked for any null values and duplicated rows, with these being removed from the dataset to ensure an accurate model was made.

4.3 SoC Estimation Models

4.3.1 Kalman Filter

The first model to be implemented was a KF, the code for which can be found in Appendix A. The model was set up as a class with several methods which allowed for easier tuning once the model was constructed. Setting up the model using object-oriented programming allowed for the model to be reused without having to re-code each iteration of the model. As discussed in **3.2.1**, the KF was used to remove noise from the OCV voltage measurements to allow for a more accurate SoC estimate to be made. The model was trained using the same variables that would be available from the BMS circuit described in **4.1**.

The parameters that were tuned for this model were: the sensor noise covariance matrix (R), the process noise covariance matrix (Q), and the time step between each prediction (dt). R represents the error associated with the measurements and adjusting this value gives more or less weighting onto the error from each measurement. Q represents the error of the prediction and again gives more or less weighting depending on the value given. Finally, dt represents the time step in between each prediction, with a smaller time step giving a more accurate model but also requiring more computational power resulting in a longer training time. The optimal values for each were found to be:

$$R = 100, \qquad Q = 0.01, \qquad dt = 0.001$$

The resulting graph once the model had been trained on the dataset, showing predicted OCV against measured, is shown in Figure 8 with the predicted value in blue and the measured values in orange. As the values for OCV showed large fluctuations, an error for this model was unable to be calculated since there was no actual known value to compare the prediction to. However, the graph suggests that the model was successful in giving an approximation of the actual OCV. Due to the inability to calculate the error, this model will not be used in any further models as it's unclear whether it would positively or negatively affect the predicted error.

Figure 8 - Graph of predicted OCV against measured

4.3.2 Random Forest Regression

The next model to be constructed was the RFR model which was used to predict the SoC of the battery at any given time, the code for this can be found in Appendix A. These estimations could be used in conjunction with the coulomb counting method to give a better estimation of the SoC. The model uses 5 cross validations in an 80/20 split between training/validation to ensure that the model is not under-estimating the error in the predicted values. The loss function used was the mean absolute error (MAE), which shows the mean magnitude of error between the predicted value and the measured value. Before tuning, the model gave a MAE of 14.19%.

The model was then run multiple times (without cross validation due to computational and time limitations), varying the number of estimators (decision trees) with each iteration to find the optimal number of estimators. The minimum MAE was found to be 1.08% with the optimal number of estimators being 52. This is a vast improvement over the un-tuned value of 14.19%. The graph of MAE against the number of estimators is shown in Figure 9. As can be seen from this graph, the MAE greatly reduces in between 2 and 52 estimators, increasing again past this point. This suggests that the model becomes over-fit to the solution if more than 52 estimators are used. Not as much confidence can be put in these results however as no cross-validation was performed when tuning and the massive difference in MAE suggests that the model is under-predicting the MAE for these results.

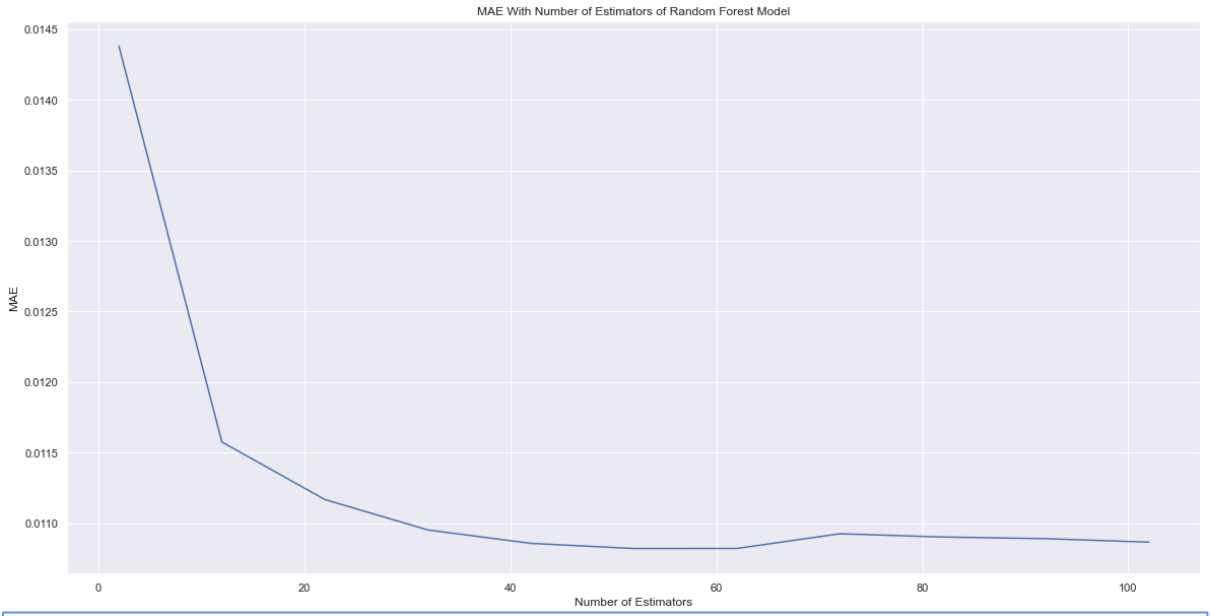


Figure 9 - Graph of MAE against number of estimators

4.3.3 Support Vector Regression

The third model to be constructed was the SVR model, the code for which can be found in Appendix A. In the same way as the RFR model, this model was used to predict SoC from the previously discussed variables. An RBF kernel was used as described in **3.2.3**. The two parameters used to tune the model were the regularisation parameter (λ) and the boundary line conditions (ε). The regularisation parameter represents the weighting given to misclassifications in the data, while the boundary line conditions have already been discussed in **3.2.3**.

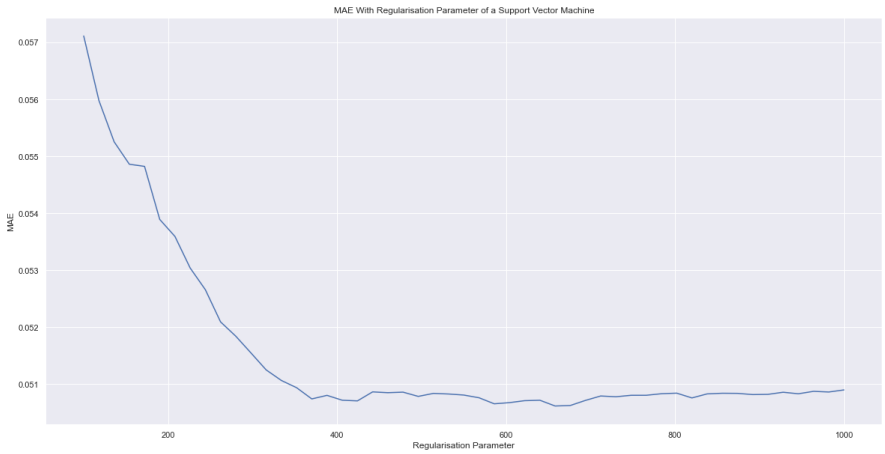


Figure 10 - Graph of MAE against Regularisation Parameter number

The model was run multiple times, varying the value for λ. This gave a minimum MAE of 5.06% with the optimal regularisation parameter number being 658. The associated graph is shown in Figure 10.

The model was then run again, this time varying the value for ε. The optimal value

for ε was found to be 0.01, with an associated MAE of 10.44%. When tuning these parameters no cross-validation was performed. The model was then changed to combine both of these optimal values and compute a MAE using cross-validation, however due to computational power and time limitations this final value was not able to be computed.

4.3.4 Neural Network

The final model that was constructed was the NN. This model was constructed using a library called Tensorflow which is a commonly used machine learning and deep learning library. This model is extremely resource intensive; with each iteration of 50 epochs taking approximately 0.25 hours. The variables used for the inputs of this model were OCV, current, battery temperature, and ambient temperature as previously discussed. The NN was trialled using both 1 and 2 hidden layers with various combinations of node numbers to attempt to find the optimal solution. The random seed generated by Tensorflow was set at a constant value of 13 to ensure reproducibility in the results. An 80/20 cross-validation split was used to ensure the model didn't under-estimate the MAE. Each trial was tested over 50 epochs until the optimal solution was found. This optimal solution was then run using 7000 epochs to find the optimal number of epochs to train the model. Training the model using 7000 epochs took approximately 35 hours of computing time when using an i5-3570K processor overclocked to 3.8GHz.

From the initial trials that were run, the model gave a much lower MAE value when 1 hidden layer was used. Trials using 2 hidden layers gave an average MAE of 17.44%, whereas trials using 1 hidden layer gave an average MAE of 9.86%. As such, 1 hidden layer was chosen to be used for the final model. Varying numbers of nodes (between 1-1000 nodes) were used to train the model to account for both of the rules mentioned in **3.2.4**. The optimal number of computational nodes in the hidden layer was found to be 4, with an associated MAE of 5.82%.

The model was then run using 1 hidden layer and 4 computational nodes across 7000 epochs, the results of which are shown in Figure 11. The optimal number of epochs was found to be 4200 with an associated MAE of 3.10%. This value included cross-validation so the confidence in the accuracy is greater than that of the previous models. As can be seen from the graph, once the epoch value went above 4200, the model massively over-fit the solution and the MAE increased to a roughly constant value of 19.80%. As this model gave the lowest MAE, it would be used as the final model for SoC estimation.

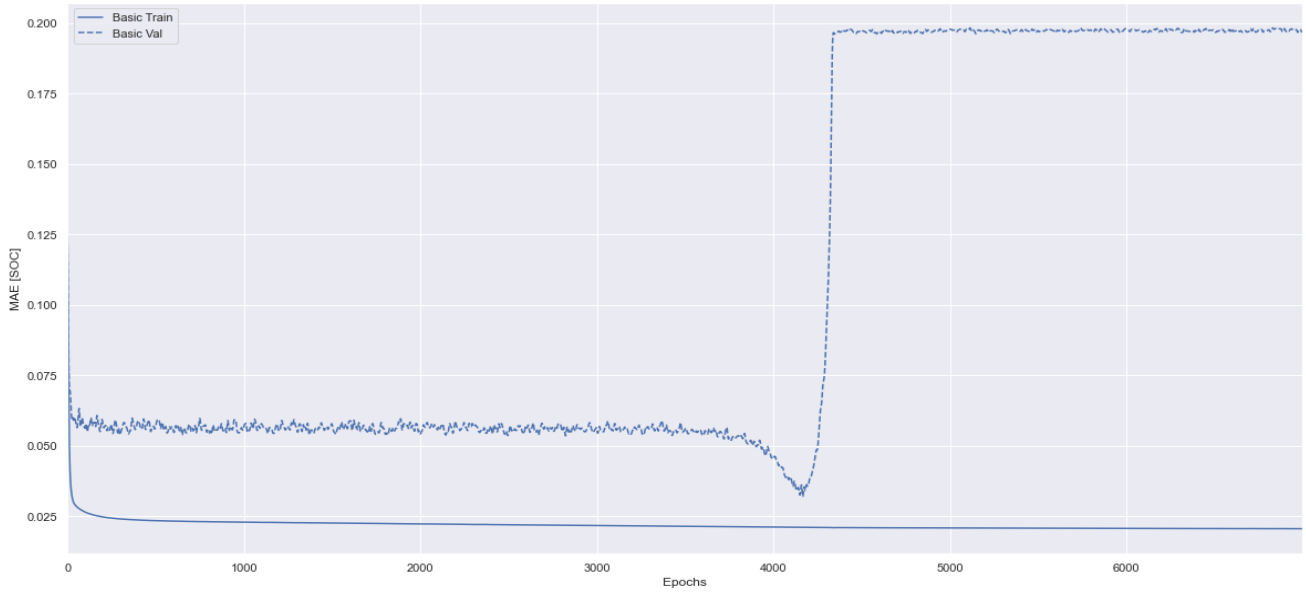


Figure 11 - Graph of MAE against Epochs for 1 hidden layer and 4 computational nodes

This final model, using 1 hidden layer, 4 computational nodes, and 4200 epochs was tested on the remaining drive cycles to validate the MAE of the model on data that the model has never seen before. This dataset included 1,126,083 data entries so the confidence in the final MAE value is extremely high. The final validated MAE for the trained model was found to be 6.96% which is higher than when the model was validated on previously seen data, although this is to be expected.

5. Evaluation

5.1 Comparison of SoC Estimation Models

The four SoC estimation models constructed for this project were implemented with varying degrees of success. The KF that was implemented gave what appeared to be a good approximation of the OCV from the measured values, however as there were no actual values to compare the estimates to, the MAE could not be calculated. As such this model will not be included in the comparison.

The lowest MAE across the three models came from the RFR model, with a tuned MAE of 1.08%. However, as the tuned version of this model was not run using cross-validation there is less confidence that the final result is an accurate representation of the MAE; the MAE would likely increase if cross-validation was performed. This also applies to the SVR model that was made. The NN was the only model that included cross-validation across all iterations and as such there is greater confidence in the result of this model. Preliminary research showed that NNs generally have better performance than the other two models with regards to SoC estimation and as such this was the model that had the most time dedicated to it. The average MAE across the two tuned versions of the SVR model without cross-validation was 7.75%. Even without cross-validation, the SVR gave worse results than the cross-validated NN and the MAE would likely increase for the SVR if cross-validation was performed. Although this was the case, the final MAE value for the SVR using both of the optimised parameters was unable to be calculated so there is no way to determine for certain if the SVR model would have been more accurate in its estimates than the NN or the RFR model.

As the tuned MAE for the RFR was 1.08%, which is significantly lower than the average tuned value for the SVR, it can be assumed that the RFR model would perform better than the SVR model if cross-validation was performed on both; although as stated this cannot be said with certainty. It is difficult to draw direct comparisons between the accuracy of these models due to the incomplete training procedure for the RFR and SVR models. When comparing the NN and the RFR, the fairest comparison would be to compare the un-tuned cross-validated MAE for the RFR with the average un-tuned cross-validated MAE for the NN using 1 hidden layer. The relative value of this for the RFR was 14.19% and the relative value for the NN was 9.86%. The NN shows a MAE that is 4.33% lower than the RFR model. This shows that the model is significantly more accurate when un-tuned. As such, the NN can be assumed to be the best model to estimate the SoC as it also gave the lowest cross-validated MAE of 6.96% and research into SoC estimation showed that NNs often perform significantly better than other models.

5.2 Improvements to the Models

If more time was available, the KF model would have had all of the parameters adjusted to give the most accurate model possible. The parameters that would be adjusted can be found in **3.2.1.**

The SVR model and RFR model would be run including cross-validation so that a direct comparison could be made between their final MAE values and the values from the NN. In addition, the NN model would be run using all the combinations of hidden layers and number of nodes across 7000 epochs to ensure that one of the combinations did not drastically improve its MAE at a larger epoch value; unfortunately this could not be done due to the computational time it took to run one iteration. The NN model could also be tested using more hidden layers, ideally testing it on up to 5 hidden layers with various node combinations, to ensure that a better solution was not missed.

The NN could also have included an LSTM (Long-Short-Term-Memory) layer, which allows for previous estimations and their error to be remembered by the NN. This would incorporate a feedback loop into the NN and greatly improve the accuracy. One journal article found used an LSTM layer and gained an error of 0.573% when predicting SoC for a li-ion battery, which suggests vast improvements to the model could be made with this addition [36]. The code for this was added, however it was far too computationally intensive to be run effectively.

5.3 Improvements to the BMS

The BMS could be improved by adding a more robust voltage sensor than the proposed voltage divider. As discussed, a voltage sensor could be used instead that connects directly to the analog input of the Arduino Uno. This would give a more accurate and precise voltage reading that would in turn improve the accuracy of the models when used.

In addition to this, the capacity of the battery could be found using a more accurate method. The proposed method involved discharging the battery at a known current and monitoring the time it takes for the battery to fully discharge. A much more accurate method would be to perform electrochemical impedance spectroscopy, if this equipment was available this would be the suggested method [39].

6. Conclusion

A Battery Monitoring System was designed to allow for monitoring of a li-ion battery when in use on the Mars Rover. This BMS would also allow for a dataset to be made that could then be used to train the Neural Network that was constructed.

This project proposed four different solutions to estimate the SoC of a li-ion battery, a Kalman filter, a Random Forest Regression model, a Support Vector Regression model, and a Neural Network. The models were assessed on their effectiveness by calculating the MAE for each model. The Neural Network gave the lowest overall cross-validated MAE of 6.96%. Other models gave fairly accurate results which could have been improved upon if more time or more computational power was available. The dataset used included over 100,000 data entries to ensure that the MAE for each model was an accurate depiction of its ability.

The best model and the final solution was concluded to be the Neural Network due to it giving the lowest cross-validated MAE value in conjunction with preliminary research on the topic. The model could have been improved by adding a Long-Short-Term-Memory layer to incorporate a feedback loop into the model. The Neural Network model can account for fluctuations in the OCV measurement without the need for a Kalman filter to be used.

References

[1] D. Linden, Handbook of batteries, Elsevier Science, 1995.

[2] C. D. W. a. C.-Y. Rahn, Battery Systems Engineering, John Wiley & Sons, 2013.

[3] M. W. a. P. B. Georg Bieker, "Electrochemical in situ investigations of SEI and dendrite formation on the lithium metal anode," *Physical Chemistry Chemical Physics,* no. 17, pp. 8670-8679, 2015.

[4] M. A. &. J.-M. Tarascon, "Building better batteries," *Nature,* vol. 451, pp. 652-657, 2008.

[5] S. S. Zhang, "A review on electrolyte additives for lithium-ion batteries," *Journal of Power Sources,* vol. 162, no. 2, pp. 1379-1394, 2006.

[6] B. M. Y. E.-E. D. Aurbach, "Investigation of Graphite-Lithium Intercalation Anodes for Li-Ion Rechargeable Batteries," *New Promising Electrochemical Systems for Rechargeable Batteries,* vol. 6, pp. 63-75, 1996.

[7] B. M. D. A. Y. C. H. Y. S. L. Yair Ein-Eli, "The dependence of the performance of Li-C intercalation anodes for Li-ion secondary batteries on the electrolyte solution composition," *Electrochimica Acta,* vol. 39, no. 17, pp. 2559-2569, 1994.

[8] J. L. C. D. D. M. S. N. D. L. W. Seong Jin An, "The state of understanding of the lithium-ion-battery graphite solid electrolyte interphase (SEI) and its relationship to formation cycling," *Carbon,* vol. 105, pp. 52-76, 2016.

[9] A. Manthiram, "A reflection on lithium-ion battery cathode chemistry," *Nature Communications,* vol. 11, 2020.

[10] D. A. Bhatt, "Lithium-ion batteries," Australian Academy of Science, June 2019. [Online]. Available: https://www.science.org.au/curious/technology-future/lithium-ion-batteries. [Accessed 2020].

[11] P. P. X. Z. G. C. J. S. C. C. Qingsong Wang, "Thermal runaway caused fire and explosion of lithium ion battery," *Journal of Power Sources,* vol. 208, pp. 210-224, 2012.

[12] P. A. P. B. B.-E. M. Fredrik Larsson, "Toxic fluoride gas emissions from lithium-ion battery fires," *Scientific Reports,* vol. 7, 2017.

[13] R. R. T. P. K. P.G. Balakrishnan, "Safety mechanisms in lithium-ion batterie," *Journal of Power Sources,* vol. 155, no. 2, pp. 401-414, 2006.

[14] K. T. Y. S. J.-i. Y. Shin-ichi Tobishima, "Lithium ion cell safety," *Journal of Power Sources,* vol. 90, no. 2, pp. 188-195, 2000.

[15] J. S. M. O. X. H. L. L. X. H. M. F. H. P. Xuning Feng, “Characterization of large format lithium ion battery exposed to extremely high temperature,” *Journal of Power Sources,* vol. 272, no. 25, pp. 457-467, 2014.

[16] K. X. T. J. S.S. Zhang, “Electrochemical impedance study on the low temperature of Li-ion batteries,” *Electrochimica Acta,* vol. 49, no. 7, pp. 1057-1061, 2004.

[17] K. X. T. J. S.S. Zhang, “The low temperature performance of Li-ion batteries,” *Journal of Power Sources,* vol. 115, no. 1, pp. 137-140, 2003.

[18] K. L. Gering, “Low-Temperature Performance Limitations of Lithium-Ion Batteries,” *The Electrochemical Society,* vol. 1, no. 25, 2006.

[19] W. H. M. P. K. L. T. Yinjiao Xing, “State of charge estimation of lithium-ion batteries using the open-circuit voltage at various ambient temperatures,” *Applied Energy,* vol. 113, pp. 106-115, 2014.

[20] Y. K. A.G. Stefanopoulou, “System-level management of rechargeable lithium-ion batteries,” in *Rechargeable Lithium Batteries,* Woodhead Publishing, 2015.

[21] The University of Texas at Dallas, “State of Charge Estimation Problem,” UT Dallas, 2017. [Online]. Available: https://labs.utdallas.edu/essl/projects/state-of-charge-estimation-problem/. [Accessed April 2020].

[22] G. B. Greg Welch, An Introduction to the Kalman Filter, Chapel Hill: University of North Carolina, 1997.

[23] R. X. X. Z. F. S. J. F. Hongwen He, “State-of-Charge Estimation of the Lithium-Ion Battery Using an Adaptive Extended Kalman Filter Based on an Improved Thevenin Model,” *IEEE Transactions on Vehicular Technology,* vol. 60, no. 4, pp. 1461-1469, 2013.

[24] X. D. B. J. Zheng Liu, “A Novel Open Circuit Voltage Based State of Charge Estimation for Lithium-Ion Battery by Multi-Innovation Kalman Filter,” *IEEE Access,* vol. 7, pp. 49432-49447, 2019.

[25] B. M. K. R. M. U. H. P. B. W. P. F. A. H. Bjoern H Menze, “A comparison of random forest and its Gini importance with standard chemometric methods for the feature selection and classification of spectral data,” *BMC Bioinformatics,* vol. 10, no. 213, 2009.

[26] M. W. Andy Liaw, “Classification and Regression by Random Forest,” *R News,* vol. 3, pp. 18-22, 2002.

[27] M. R. Segal, “Machine Learning Benchmarks and Random Forest Regression,” *UCSF: Center for Bioinformatics and Molecular Biostatistics,* vol. 6, 2004.

[28] Z. C. J. C. Y. W. F. Z. Chuanjiang Li, “The lithium-ion battery state-of-charge estimation using random forest regression,” *Prognostics and System Health Management,* vol. 5, 2014.

[29] B. S. Alex J. Smola, “A tutorial on support vector regression,” *Statistics and Computing,* vol. 14, pp. 199-222, 2004.

[30] J. H. X. C. X. W. J.N.Hu, “State-of-charge estimation for battery management system using optimized support vector machine for regression,” *Journal of Power Sources,* vol. 269, pp. 682-693, 2014.

[31] M. J. L. Orr, Introduction to Radial Basis Function networks, Edinburgh: Centre for Cognitive Science, 1996.

[32] S. S. K. M. S. Malkhandi, “Estimation of state of charge of lead acid battery using radial basis function,” in *IECON'01. 27th Annual Conference of the IEEE Industrial Electronics Society*, Denver, 2001.

[33] H. B. D. M. B. Martin T. Hagan, Neural Network Design, Computer Science, 1995.

[34] S. S.-S. O. S. Roi Livni, “On the Computational Efficiency of Training Neural Networks,” in *Advances in Neural Information Processing Systems*, 2014.

[35] C. L. Xu S, “A novel approach for determining the optimal number of hidden layer neurons for FNN’s and its application in data mining,” in *5th International Conference on Information Technology and Applications*, Cairns, 2008.

[36] P. J. K. M. P. R. A. A. E. E. Chemali, “Long Short-Term Memory Networks for Accurate State-of-Charge Estimation of Li-ion Batteries,” *IEEE Transactions on Industrial Electronics,* vol. 65, no. 8, pp. 6730-6739, 2018.

[37] J. Blom, “Voltage Dividers,” Sparkfun, 2008. [Online]. Available: https://learn.sparkfun.com/tutorials/voltage-dividers/all. [Accessed 28 April 2020].

[38] P. Kollmeyer, “Panasonic 18650PF Li-ion Battery Data,” 21 June 2018. [Online]. Available: https://data.mendeley.com/datasets/wykht8y7tg/1. [Accessed 1 April 2020].

[39] A. E. T. K. N. L. O. F. E. J. L. Jespersen, “Capacity Measurements of Li-Ion Batteries,” *World Electric Vehicle Journal,* vol. 3, pp. 127-133, 2009.

Appendices
Appendix A – Code for the model

Imports

This section imports all the needed libraries for the data analysis.

Numpy: A library written in C with a Python wrapper to give much faster computation. Allows for easy manipulation of arrays and lists with added functionality, also allows for the use of a Kalman Filter.

Pandas: Another library that is written in C using a Python wrapper. An extremely important library for data analysis, used for working with tabular data. Includes intuitive and additional functions that interface with **Numpy**

Matplotlib: A library used to visualise data.

Seaborn: An extension of Matplotlib that uses Matplotlib's backend. Again, used for visualising data and graphing.

Scipy: A library useful for process signalling and statistical analysis. Can be used to interpret data from electronic sensors.

Sklearn: A Machine learning library for Python that includes multiple useful loss functions such as Mean absolute error (MAE) and Mean squared error (MSE). Can be used to split data sets for testing and validating and allows for easy cross validation in machine learning algorithms. Includes the Random Forest Regression model used.

In [1]:

```
# Linear Algebra
import numpy as np
from filterpy.kalman import KalmanFilter

# Data Processing
import pandas as pd
from filterpy.common import Q_discrete_white_noise

# Data Visualization
import seaborn as sns
import matplotlib.pyplot as plt

# Stats
from scipy import stats

# Algorithms
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.metrics import mean_absolute_error

# Regressors
from sklearn.ensemble import RandomForestRegressor
from sklearn.svm import SVR

import os

# Set random seed to be constant for reproducable results
np.random.seed(0)

sns.set()  # Stylises graphs
```

Importing Data

This section imports the dataset that was converted from .mat format to .csv to allow it to be used with Python. The data had to be reformatted before this point so that is easily usable in the models.

The data is visualised in a table and checked for any null values which could cause errors in the results. The data type for each column is also checked to make sure it is consistent. The integer value for Chamber Temp is usable as Python is able to work with integer and float values together by automatically converting between the two.

In [2]:

```
df = pd.read_csv('./csv-data/03-18-17_02.17 25degC_Cycle_1_Pan18650PF.csv')
```

In [3]:

```
df.head()
```

Out[3]:

	Voltage	Current	Ah	Wh	Power	Battery Temp	Time	Chamber Temp
0	4.14585	-1.81290	-0.00000	-0.00000	-7.516011	21.781981	0.000000	23
1	4.10532	-1.83249	-0.00005	-0.00020	-7.522958	21.781981	0.095996	23
2	4.08666	-1.85046	-0.00010	-0.00042	-7.562201	21.781981	0.202001	23
3	4.08087	-1.85781	-0.00015	-0.00062	-7.581481	21.781981	0.297997	23
4	4.07765	-1.86353	-0.00021	-0.00084	-7.598823	21.781981	0.403005	23

In [4]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 109641 entries, 0 to 109640
Data columns (total 8 columns):
 #   Column        Non-Null Count   Dtype  
---  ------        --------------   -----  
 0   Voltage       109641 non-null  float64
 1   Current       109641 non-null  float64
 2   Ah            109641 non-null  float64
 3   Wh            109641 non-null  float64
 4   Power         109641 non-null  float64
 5   Battery Temp  109641 non-null  float64
 6   Time          109641 non-null  float64
 7   Chamber Temp  109641 non-null  int64  
dtypes: float64(7), int64(1)
memory usage: 6.7 MB
```

Missing Values

The dataset is checked once again to ensure that there are no null values (missing data points) across the entire dataset.

In [88]:
```
df.isnull().sum() * 100 / df.shape[0]
```

Out[88]:
```
Voltage         0.0
Current         0.0
Ah              0.0
Wh              0.0
Power           0.0
Battery Temp    0.0
Time            0.0
Chamber Temp    0.0
dtype: float64
```

Duplicates

This function checks for any duplicates in the dataset and displays the total amount of duplicates. Duplicates in the dataset may indicate data-logging errors and so should be removed from the dataset to ensure consistent and accurate results. As there are only 3 duplicates in a large dataset they will have minimal impact on the final result of the analysis however they have still been removed from the dataset.

In [89]:
```
print(f'Duplicates: {df.duplicated().sum()}')
```

```
Duplicates: 3
```

The code below removes the duplicated data entries from the dataset.

In [5]:
```
df = df.drop_duplicates()
```

Capacity

The typical capcity of this battery is **2784mAh**. This function converts the **Ah** measurement into a relative capacity by determining the capacity lost from the battery and dividing this by the known maximum capacity of the battery.

In [6]:
```
df['Capacity'] = 1 - (-df['Ah'] / 2.784)
```

In [95]:
```
df.head()
```

Out[95]:

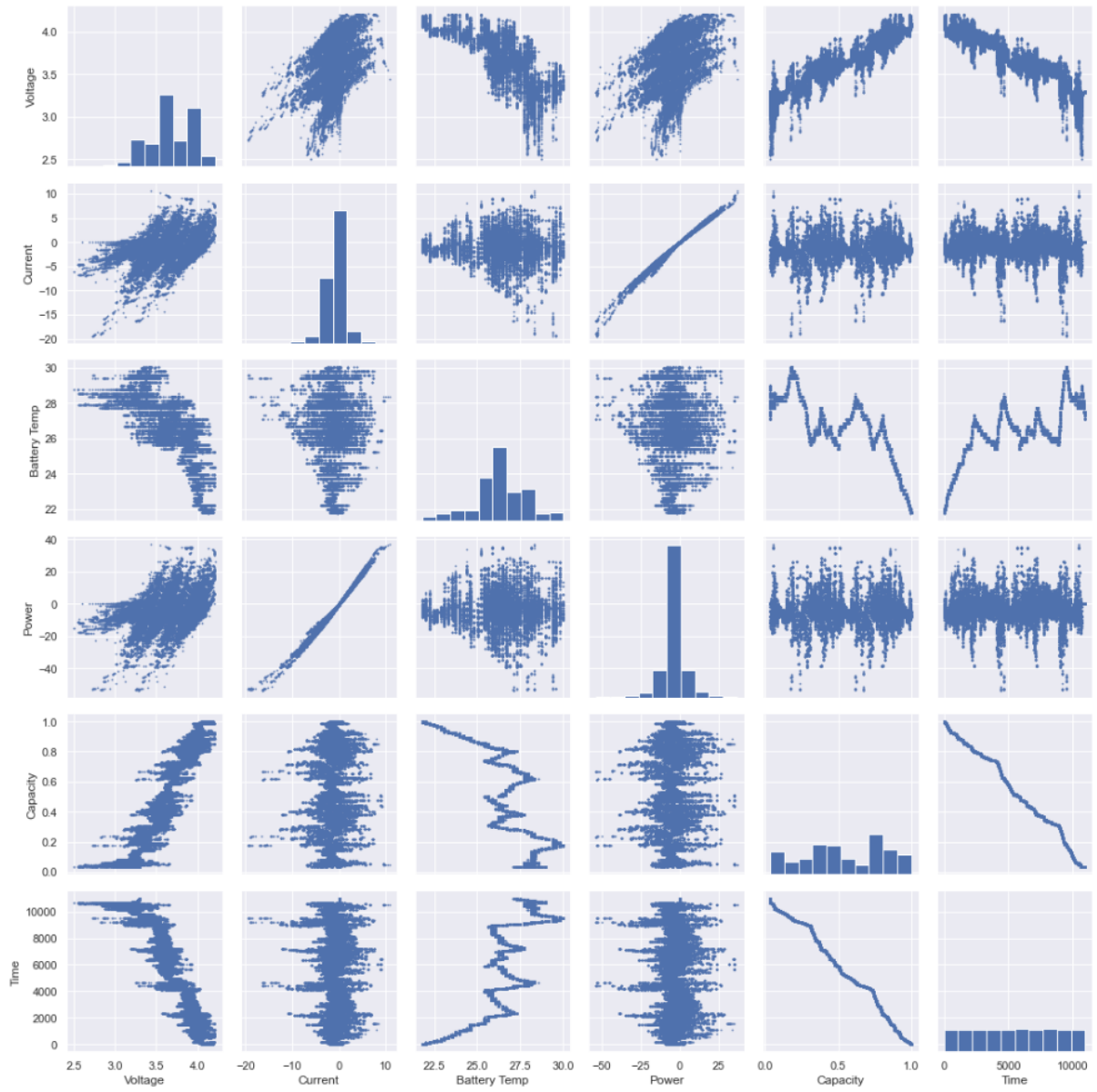
	Voltage	Current	Ah	Wh	Power	Battery Temp	Time	Chamber Temp	Capacity
0	4.14585	-1.81290	-0.00000	-0.00000	-7.516011	21.781981	0.000000	23	1.000000
1	4.10532	-1.83249	-0.00005	-0.00020	-7.522958	21.781981	0.095996	23	0.999982
2	4.08666	-1.85046	-0.00010	-0.00042	-7.562201	21.781981	0.202001	23	0.999964
3	4.08087	-1.85781	-0.00015	-0.00062	-7.581481	21.781981	0.297997	23	0.999946
4	4.07765	-1.86353	-0.00021	-0.00084	-7.598823	21.781981	0.403005	23	0.999925

Visualisations

This section visualises the data to show any obvious trends between parameters. This allows easier understanding of the data and can be useful when tuning the statistical models to determine which variables are most important.

In [96]:
```
sns.pairplot(
    df[['Voltage', 'Current', 'Battery Temp', 'Power', 'Capacity', 'Time']],
    diag_kind='hist', palette='bright',
    plot_kws={'facecolor': 'b', 'edgecolor': 'b', 's': 1}
)

plt.tight_layout()
```



Time Period of Data Collection

This section of the code converts the time steps in the dataset into a total run time for the data set.

In [97]:
```
print(f"Period of data collection: {round(max(df['Time']) / (60 ** 2), 2)} hrs")
```

```
Period of data collection: 3.05 hrs
```

Machine Learning

This part of the code is where the machine learning algorithms are set up. Four different machine learning methods have been used in this program, a **Kalman filter**, **Random forest regression model**, **Support vector regression model**, and a **Recurrent Neural Network model**.

In [7]:
```
X = df.copy()
y = X['Capacity']

X = X.drop(columns=['Time', 'Wh', 'Ah', 'Capacity'])

X_train, X_valid, y_train, y_valid = train_test_split(X, y, train_size=0.01, random_state=0)
```

Kalman Filter

The Kalman filter is set up as a class with several methods. This allows the parameters to be easily tuned once the model has been run, allowing for improved results. The Kalman Filter is a filtering method used to smooth out data that comes from a noisy source (such as the electronic readings for voltage in this dataset). This allows for a good approximation of the actual values despite the fluctuation in data.

In [8]:
```
class BatteryCapacityKalmanFilter():
    def __init__(self, X, y_measured, P, R, Q, dt):
        self.X = X
        self.y_measured = y_measured
        self.P = P
        self.R = R
        self.Q = Q
        self.dt = dt

    @property
    def dim_x(self):
        return self.X.shape[1]

    def capacity_filter(self):
        kf = KalmanFilter(dim_x=self.dim_x, dim_z=1)

        # Sets up a numpy array with Voltage, Current, Power, Bat Temp, and Chamber Temp
        kf.x = np.array(self.X.loc[0])

        # State transition matrix
        kf.F = np.array(
            [[1, self.dt],
            [0,  1]]
        )
        # Measurement function
        kf.H = np.array([[1, 0]])
        # Measurement uncertainty
        kf.R *= self.R

        if np.isscalar(self.P):
            # Covariance matrix
            kf.P *= self.P
        else:
            kf.P[:] = self.P
        if np.isscalar(self.Q):
            kf.Q = Q_discrete_white_noise(dim=self.dim_x, dt=self.dt, var=self.Q)
        else:
            kf.Q[:] = self.Q
        return kf

    def run(self):
        # Creates the Kalman filter
        kf = self.capacity_filter()

        # Run the kalman filter and store the results in an array
        x, cov = [], []

        for y in self.y_measured:
            kf.predict()
            kf.update(y)

            x.append(kf.x[0])
            cov.append(kf.P)

        x, cov = np.array(x), np.array(cov)
        return x, cov
```

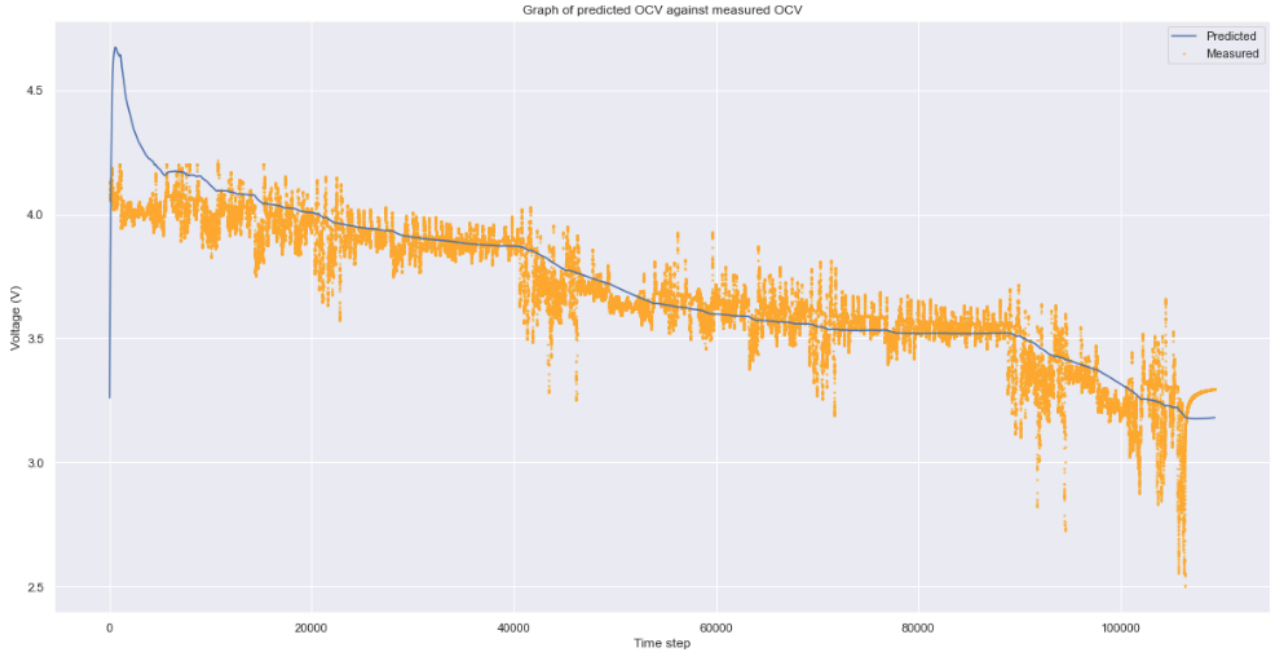
In [9]:
```
P = np.diag([1, 20])
batt_kf = BatteryCapacityKalmanFilter(
    X=X[['Current', 'Power']],
    y_measured=X['Voltage'],
    P=P, R=100, Q=0.01, dt=0.001
)
x, cov = batt_kf.run()
```

In [10]:
```
# Visualising the data, with measured voltage values shown in orange and the Kalman filter prediction shown in blue.
plt.figure(figsize=(20, 10))

plt.plot(x[300:], label='Predicted')
plt.scatter(
    range(len(X[300:]['Voltage'])), X[300:]['Voltage'],
    label='Measured', c='orange', s=1
)
plt.title('Graph of predicted OCV against measured OCV')
plt.xlabel('Time step')
plt.ylabel('Voltage (V)')

# plt.xticks(np.linspace(min(y), max(y)), )
plt.legend()

plt.show()
```



Random Forest

This section sets up a Random forest regression model that can be used to estimate the **Ah** measurement from the other known parameters. These estimations can be used in conjunction with measurements from a coulomb counter to give much more accurate results allowing for better estimation of the State of Charge (SoC) of the battery.

The model uses 5 cross validations in an 80/20 split between training/validation. This ensures that the model is not over estimating it's ability to predict the **Ah** value thus ensuring there isn't false confidence in inaccurate results.

The mean of all 5 cross validations is then calculated and taken to be the MAE of the model.

The model is then tuned by varying the number of estimators in the Random forest model. The data is graphed to show the MAE across the different parameter numbers and from this data the optimal number of estimators can be determined and then used in the final model to ensure the best accuracy possible.

In [102]:
```
model_scores = -cross_val_score(
   estimator=RandomForestRegressor(),
   X=X, y=y,
   scoring='neg_mean_absolute_error', cv=5
 )

print(f'Model Scores: {model_scores}')
```

```
Model Scores: [0.16008808 0.0542119  0.09649997 0.06046793 0.33826397]
```

In [103]:
```
print(f'SoC MAE: {round(model_scores.mean() * 100, 2)}%')
```

```
SoC MAE: 14.19%
```

In [104]:
```
scores = []
n_estimators = range(2, 103, 10)

for n_estimator in n_estimators:
    rf = RandomForestRegressor(n_estimators=n_estimator, random_state=0)
    rf.fit(X_train, y_train)

    score = mean_absolute_error(rf.predict(X_valid), y_valid)
    scores.append(score)
```

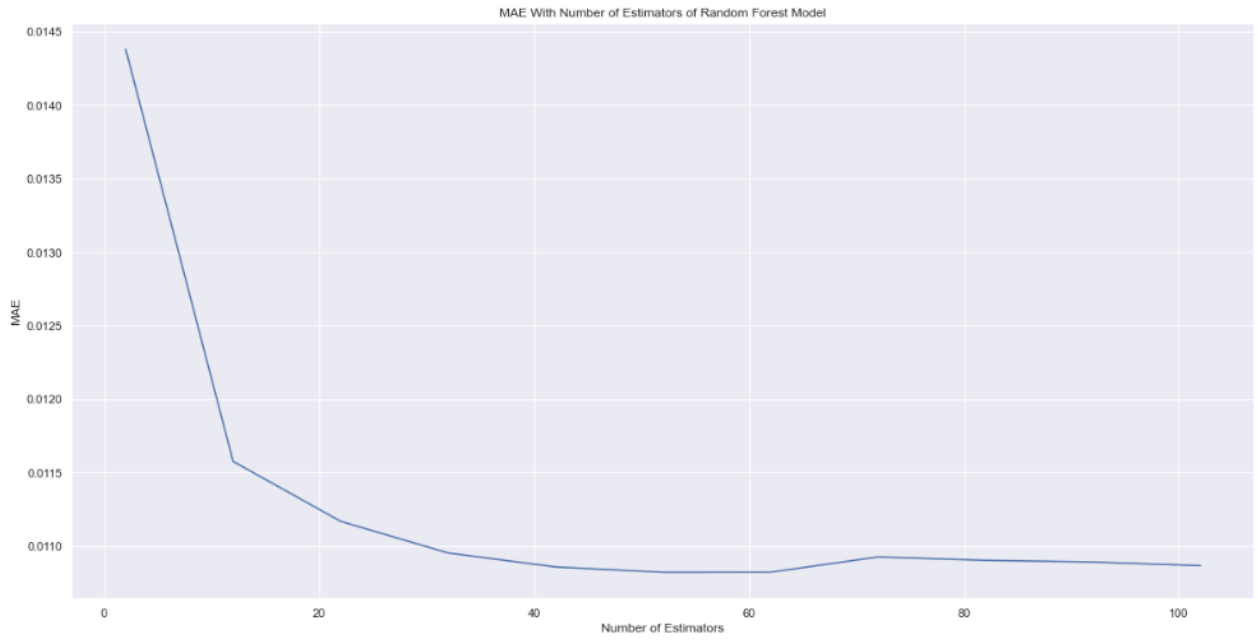
In [105]:
```
print(f'Minimum MAE: {min(scores) * 100}%')
print(f'Optimal Number of Estimators: {(10*np.argmin(scores)) + 2}')
```

```
Minimum MAE: 1.0822612665241502%
Optimal Number of Estimators: 52
```

In [106]:
```
plt.figure(figsize=(20, 10))

plt.plot(n_estimators[:100], scores)
plt.title('MAE With Number of Estimators of Random Forest Model')
plt.xlabel('Number of Estimators')
plt.ylabel('MAE')

plt.show()
```



Support Vector Machines

This section sets up and runs a Support Vector Machine (SVM) model, more specifically a Support Vector Regression (SVR) model.

There are two parameters for this model that can be tuned to improve the accuracy of the results, the Regularisation parameter, and the ε parameter as shown below.

Regularisation Parameter

The data for the Regularisation parameter is again graphed to show the optimal parameter number for this model. As can be seen from the graph, this model gives superior results to the previous Random forest regression model even with only one parameter tuned.

In [107]:
```
scores = []
reg_params = np.linspace(100, 1000, 51)

for C in reg_params:
    svm = SVR(kernel='rbf', C=C)
    svm.fit(X_train, y_train)

    score = mean_absolute_error(svm.predict(X_valid), y_valid)
    scores.append(score)
```

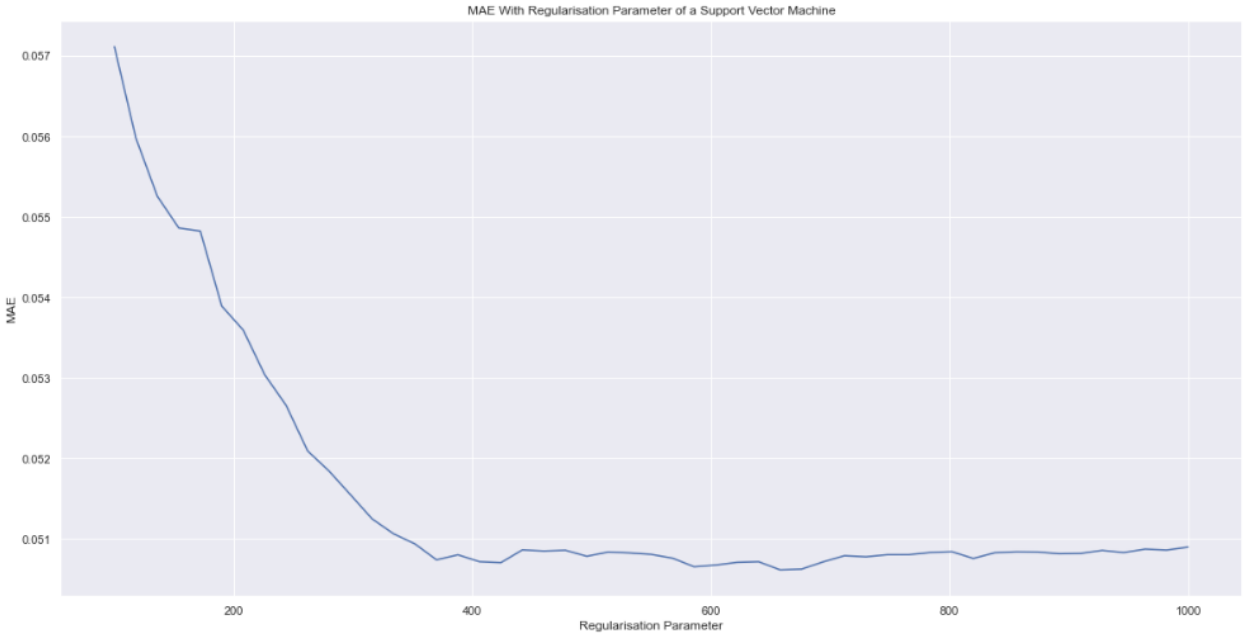
In [108]:
```
print(f'Minimum MAE: {min(scores) * 100}%')
print(f'Optimal Regulator Parameter number: {round(reg_params[np.argmin(scores)], 0)}')
```

```
Minimum MAE: 5.061514315657602%
Optimal Regulator Parameter number: 658.0
```

In [109]:
```
plt.figure(figsize=(20, 10))

plt.plot(reg_params, scores)
plt.title('MAE With Regularisation Parameter of a Support Vector Machine')
plt.xlabel('Regularisation Parameter')
plt.ylabel('MAE')

plt.show()
```



ε Parameter

This section graphs the varying values of ε as a parameter against MAE, again to allow the best parameters to be chosen for the model. Tuning the ε parameter, before combining with the tuned Regularisation parameter, gives a minimum MAE of 0.1044.

In [157]:
```
scores = []
e_values = np.linspace(1e-10, 1e-1, 51)

for e in e_values:
    svm = SVR(kernel='rbf', epsilon=e)
    svm.fit(X_train, y_train)

    score = mean_absolute_error(svm.predict(X_valid), y_valid)
    scores.append(score)
```

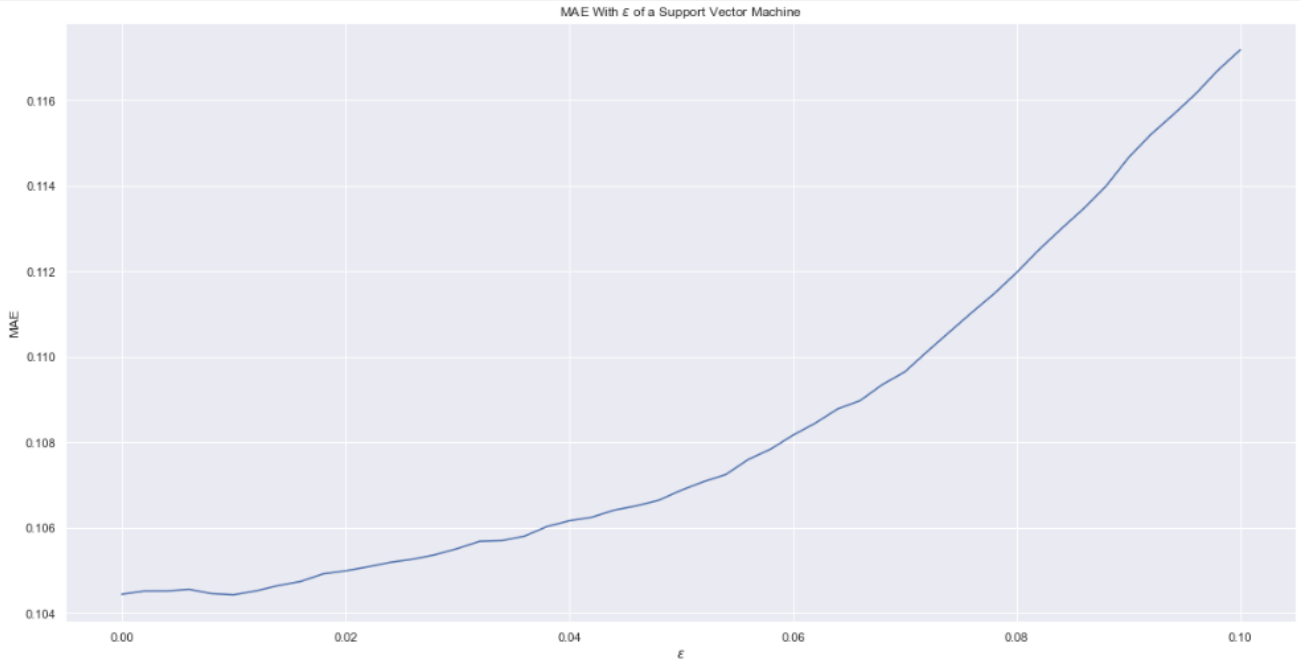
In [160]:
```
print(f'Minimum MAE: {min(scores) * 100}%')
print(f'Optimal Epsilon value: {round(e_values[np.argmin(scores)], 4)}')
```

```
Minimum MAE: 10.44287628115331%
Optimal Epsilon value: 0.01
```

In [159]:
```
plt.figure(figsize=(20, 10))

plt.plot(e_values, scores)
plt.title('MAE With $\epsilon$ of a Support Vector Machine')
plt.xlabel('$\epsilon$')
plt.ylabel('MAE')

plt.show()
```



Combine

The two tuned parameters are then passed into a new model that includes cross validation to ensure the model is giving the most accurate results possible. Unfortunately due to the computing power available, this was unable to be run effectively and so has been commented out to allow the remaining code to run.

In [30]:
```
#svm = SVR(kernel='rbf', C=658, epsilon=0.0102)

#model_scores = -cross_val_score(
#    estimator=svm,
#    X=X, y=y,
#    scoring='neg_mean_absolute_error', cv=5, n_jobs=7
#     )

#print(f'Trained SVM Score: {model_scores.mean}')
```

Neural Network

This section sets up a Neural Network (NN) using a new library called Tensorflow. Tensorflow is a commonly used machine learning and deep learning library that can use the GPU in the computer to vastly improve computing times (Provided a powerful enough GPU is available).

This model is extremely resource intensive compared to the previous three models, with each iteration taking approximately 6 hours. If more time was available a Long-Short-Term-Memory (LSTM) layer would be added to the NN to improve the accuracy, this has been coded for but again, due to the computing power available has been commented out.

The NN was trialled using 1 and 2 hidden layers, with varying amounts of nodes on each iteration. The optimal solution was found to be 1 hidden layer with 4 computational nodes which is what has been shown below. This produced a MAE of 3.8%.

The amount of epochs in the NN was set to a constant 50 for each trial, however this could be massively increased and a large increase in accuracy would be expected if this was done. Increasing the amount of epochs increases the computational time per iteration and thus was set at a constant 50. In an ideal scenario, each iteration would be run for 7000 epochs and the minimum MAE across all epochs would be taken for each iteration.

The MAE and MSE are graphed below for easy comparison between each iteration.

In [31]:
```
import tensorflow as tf
tf.random.set_seed(13)
from tensorflow.keras import layers
from tensorflow.keras.layers import LSTM

import tensorflow_docs as tfdocs
import tensorflow_docs.plots
import tensorflow_docs.modeling
```

In [32]:
```
def build_model():
    model = tf.keras.Sequential()
    # Add an Embedding layer expecting input vocab of size 1000, and
    # output embedding dimension of size 64.
#     model.add(layers.Embedding(input_dim=5, output_dim=1))

     # Add a LSTM layer with 128 internal units.
#     model.add(layers.LSTM(128))

     # Add a Dense layer with 10 units.
#     model.add(layers.Dense(10))
    model = tf.keras.Sequential([
        layers.Dense(4, activation='relu', input_shape=[len(X.keys())]),
        layers.Dense(1)
    ])


    optimizer = tf.keras.optimizers.RMSprop(0.001)

    model.compile(
        loss='mse', optimizer=optimizer, metrics=['mae', 'mse']
    )

    return model
```

In [33]:
```
model = build_model()
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 4)                 24
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 5
=================================================================
Total params: 29
Trainable params: 29
Non-trainable params: 0
_________________________________________________________________
```

In [34]:
```
EPOCHS = 7000

with tf.device('/device:CPU:0'):
    history = model.fit(
        X, y,
        epochs=EPOCHS, validation_split=0.2, verbose=0,
        callbacks=[tfdocs.modeling.EpochDots()]
    )
```

```
Epoch: 0, loss:0.2240,  mae:0.2411,  mse:0.2240,  val_loss:0.0455,  val_mae:0.1879,  val_mse:0.0455,
....................................................................................................
Epoch: 100, loss:0.0011,  mae:0.0264,  mse:0.0011,  val_loss:0.0081,  val_mae:0.0513,  val_mse:0.0081,
....................................................................................................
Epoch: 200, loss:0.0010,  mae:0.0247,  mse:0.0010,  val_loss:0.0092,  val_mae:0.0522,  val_mse:0.0092,
....................................................................................................
Epoch: 300, loss:0.0009,  mae:0.0240,  mse:0.0009,  val_loss:0.0080,  val_mae:0.0527,  val_mse:0.0080,
....................................................................................................
Epoch: 400, loss:0.0009,  mae:0.0236,  mse:0.0009,  val_loss:0.0088,  val_mae:0.0510,  val_mse:0.0088,
....................................................................................................
Epoch: 500, loss:0.0009,  mae:0.0233,  mse:0.0009,  val_loss:0.0081,  val_mae:0.0556,  val_mse:0.0081,
....................................................................................................
Epoch: 600, loss:0.0009,  mae:0.0232,  mse:0.0009,  val_loss:0.0089,  val_mae:0.0512,  val_mse:0.0089,
....................................................................................................
Epoch: 700, loss:0.0009,  mae:0.0231,  mse:0.0009,  val_loss:0.0088,  val_mae:0.0515,  val_mse:0.0088,
....................................................................................................
Epoch: 800, loss:0.0009,  mae:0.0229,  mse:0.0009,  val_loss:0.0081,  val_mae:0.0538,  val_mse:0.0081,
....................................................................................................
Epoch: 900, loss:0.0009,  mae:0.0229,  mse:0.0009,  val_loss:0.0081,  val_mae:0.0577,  val_mse:0.0081,
....................................................................................................
Epoch: 1000, loss:0.0009,  mae:0.0229,  mse:0.0009,  val_loss:0.0084,  val_mae:0.0532,  val_mse:0.0084,
....................................................................................................
Epoch: 1100, loss:0.0008,  mae:0.0228,  mse:0.0008,  val_loss:0.0082,  val_mae:0.0596,  val_mse:0.0082,
....................................................................................................
Epoch: 1200, loss:0.0008,  mae:0.0227,  mse:0.0008,  val_loss:0.0086,  val_mae:0.0515,  val_mse:0.0086,
....................................................................................................
Epoch: 1300, loss:0.0008,  mae:0.0226,  mse:0.0008,  val_loss:0.0083,  val_mae:0.0533,  val_mse:0.0083,
....................................................................................................
Epoch: 1400, loss:0.0008,  mae:0.0226,  mse:0.0008,  val_loss:0.0082,  val_mae:0.0582,  val_mse:0.0082,
....................................................................................................
Epoch: 1500, loss:0.0008,  mae:0.0225,  mse:0.0008,  val_loss:0.0082,  val_mae:0.0522,  val_mse:0.0082,
....................................................................................................
Epoch: 1600, loss:0.0008,  mae:0.0224,  mse:0.0008,  val_loss:0.0082,  val_mae:0.0551,  val_mse:0.0082,
....................................................................................................
Epoch: 1700, loss:0.0008,  mae:0.0223,  mse:0.0008,  val_loss:0.0081,  val_mae:0.0595,  val_mse:0.0081,
....................................................................................................
Epoch: 1800, loss:0.0008,  mae:0.0224,  mse:0.0008,  val_loss:0.0083,  val_mae:0.0538,  val_mse:0.0083,
....................................................................................................
Epoch: 1900, loss:0.0008,  mae:0.0223,  mse:0.0008,  val_loss:0.0084,  val_mae:0.0532,  val_mse:0.0084,
....................................................................................................
Epoch: 2000, loss:0.0008,  mae:0.0222,  mse:0.0008,  val_loss:0.0081,  val_mae:0.0536,  val_mse:0.0081,
....................................................................................................
```

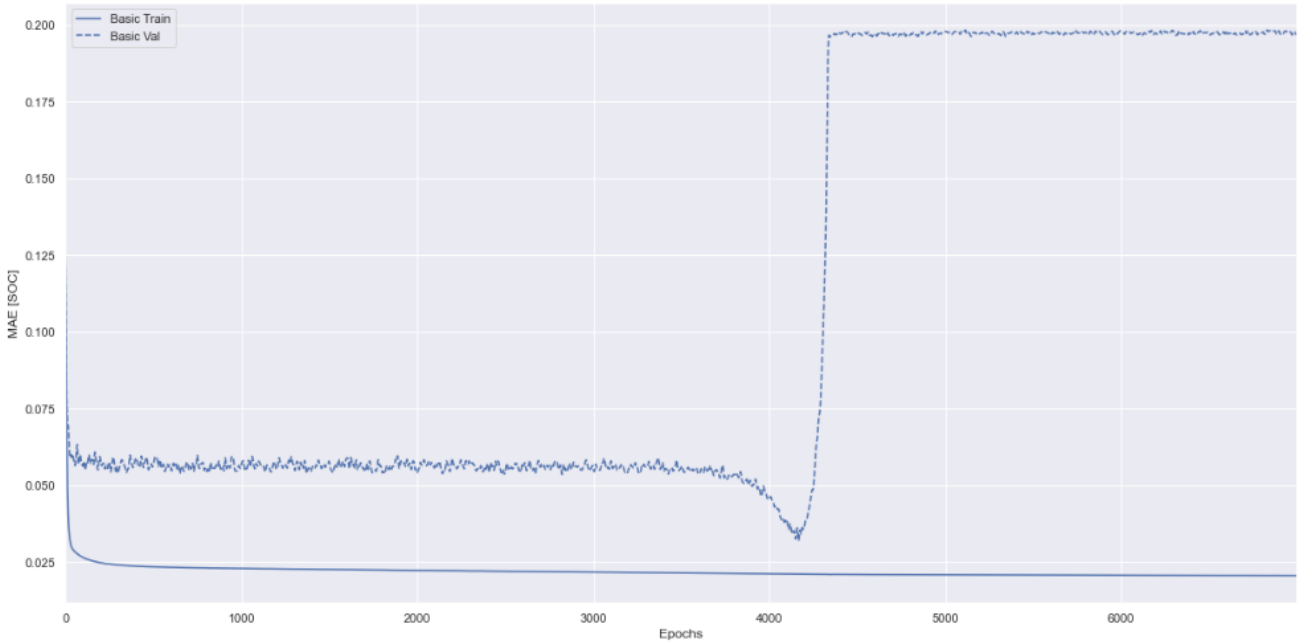
```
Epoch: 2300, loss:0.0008,  mae:0.0219,  mse:0.0008,  val_loss:0.0081,  val_mae:0.0552,  val_mse:0.0081,
.....
Epoch: 2400, loss:0.0008,  mae:0.0219,  mse:0.0008,  val_loss:0.0084,  val_mae:0.0649,  val_mse:0.0084,
.....
Epoch: 2500, loss:0.0008,  mae:0.0219,  mse:0.0008,  val_loss:0.0085,  val_mae:0.0525,  val_mse:0.0085,
.....
Epoch: 2600, loss:0.0008,  mae:0.0219,  mse:0.0008,  val_loss:0.0083,  val_mae:0.0536,  val_mse:0.0083,
.....
Epoch: 2700, loss:0.0008,  mae:0.0218,  mse:0.0008,  val_loss:0.0087,  val_mae:0.0515,  val_mse:0.0087,
.....
Epoch: 2800, loss:0.0008,  mae:0.0217,  mse:0.0008,  val_loss:0.0082,  val_mae:0.0568,  val_mse:0.0082,
.....
Epoch: 2900, loss:0.0008,  mae:0.0217,  mse:0.0008,  val_loss:0.0086,  val_mae:0.0524,  val_mse:0.0086,
.....
Epoch: 3000, loss:0.0008,  mae:0.0216,  mse:0.0008,  val_loss:0.0080,  val_mae:0.0551,  val_mse:0.0080,
.....
Epoch: 3100, loss:0.0008,  mae:0.0216,  mse:0.0008,  val_loss:0.0084,  val_mae:0.0613,  val_mse:0.0084,
.....
Epoch: 3200, loss:0.0008,  mae:0.0216,  mse:0.0008,  val_loss:0.0082,  val_mae:0.0556,  val_mse:0.0082,
.....
Epoch: 3300, loss:0.0008,  mae:0.0214,  mse:0.0008,  val_loss:0.0087,  val_mae:0.0519,  val_mse:0.0087,
.....
Epoch: 3400, loss:0.0007,  mae:0.0214,  mse:0.0007,  val_loss:0.0081,  val_mae:0.0539,  val_mse:0.0081,
.....
Epoch: 3500, loss:0.0007,  mae:0.0213,  mse:0.0007,  val_loss:0.0081,  val_mae:0.0579,  val_mse:0.0081,
.....
Epoch: 3600, loss:0.0007,  mae:0.0213,  mse:0.0007,  val_loss:0.0073,  val_mae:0.0565,  val_mse:0.0073,
.....
Epoch: 3700, loss:0.0007,  mae:0.0212,  mse:0.0007,  val_loss:0.0066,  val_mae:0.0561,  val_mse:0.0066,
.....
Epoch: 3800, loss:0.0007,  mae:0.0212,  mse:0.0007,  val_loss:0.0058,  val_mae:0.0488,  val_mse:0.0058,
.....
Epoch: 3900, loss:0.0007,  mae:0.0212,  mse:0.0007,  val_loss:0.0048,  val_mae:0.0542,  val_mse:0.0048,
.....
Epoch: 4000, loss:0.0007,  mae:0.0211,  mse:0.0007,  val_loss:0.0036,  val_mae:0.0501,  val_mse:0.0036,
.....
Epoch: 4100, loss:0.0007,  mae:0.0211,  mse:0.0007,  val_loss:0.0024,  val_mae:0.0416,  val_mse:0.0024,
.....
Epoch: 4200, loss:0.0007,  mae:0.0210,  mse:0.0007,  val_loss:0.0014,  val_mae:0.0310,  val_mse:0.0014,
.....
Epoch: 4300, loss:0.0007,  mae:0.0209,  mse:0.0007,  val_loss:0.0113,  val_mae:0.0918,  val_mse:0.0113,
.....
Epoch: 4400, loss:0.0007,  mae:0.0208,  mse:0.0007,  val_loss:0.0466,  val_mae:0.1960,  val_mse:0.0466,
.....
Epoch: 4500, loss:0.0007,  mae:0.0209,  mse:0.0007,  val_loss:0.0470,  val_mae:0.1979,  val_mse:0.0470,
.....
Epoch: 4600, loss:0.0007,  mae:0.0208,  mse:0.0007,  val_loss:0.0466,  val_mae:0.1960,  val_mse:0.0466,
.....
Epoch: 4700, loss:0.0007,  mae:0.0208,  mse:0.0007,  val_loss:0.0472,  val_mae:0.1988,  val_mse:0.0472,
.....
Epoch: 4800, loss:0.0007,  mae:0.0208,  mse:0.0007,  val_loss:0.0458,  val_mae:0.1937,  val_mse:0.0458,
.....
Epoch: 4900, loss:0.0007,  mae:0.0208,  mse:0.0007,  val_loss:0.0471,  val_mae:0.1984,  val_mse:0.0471,
.....
Epoch: 5000, loss:0.0007,  mae:0.0207,  mse:0.0007,  val_loss:0.0464,  val_mae:0.1962,  val_mse:0.0464,
.....
Epoch: 5100, loss:0.0007,  mae:0.0208,  mse:0.0007,  val_loss:0.0473,  val_mae:0.1982,  val_mse:0.0473,
.....
Epoch: 5200, loss:0.0007,  mae:0.0207,  mse:0.0007,  val_loss:0.0476,  val_mae:0.1993,  val_mse:0.0476,
.....
Epoch: 5300, loss:0.0007,  mae:0.0207,  mse:0.0007,  val_loss:0.0470,  val_mae:0.1973,  val_mse:0.0470,
.....
Epoch: 5400, loss:0.0007,  mae:0.0207,  mse:0.0007,  val_loss:0.0468,  val_mae:0.1963,  val_mse:0.0468,
.....
Epoch: 5500, loss:0.0007,  mae:0.0206,  mse:0.0007,  val_loss:0.0466,  val_mae:0.1962,  val_mse:0.0466,
.....
Epoch: 5600, loss:0.0007,  mae:0.0206,  mse:0.0007,  val_loss:0.0468,  val_mae:0.1967,  val_mse:0.0468,
.....
Epoch: 5700, loss:0.0007,  mae:0.0207,  mse:0.0007,  val_loss:0.0466,  val_mae:0.1958,  val_mse:0.0466,
.....
Epoch: 5800, loss:0.0007,  mae:0.0206,  mse:0.0007,  val_loss:0.0474,  val_mae:0.1985,  val_mse:0.0474,
.....
Epoch: 5900, loss:0.0007,  mae:0.0206,  mse:0.0007,  val_loss:0.0470,  val_mae:0.1976,  val_mse:0.0470,
.....
Epoch: 6000, loss:0.0007,  mae:0.0206,  mse:0.0007,  val_loss:0.0472,  val_mae:0.1979,  val_mse:0.0472,
.....
Epoch: 6100, loss:0.0007,  mae:0.0206,  mse:0.0007,  val_loss:0.0474,  val_mae:0.1986,  val_mse:0.0474,
```

```
....................................................................................................
Epoch: 6200, loss:0.0007,  mae:0.0206,  mse:0.0007,  val_loss:0.0468,  val_mae:0.1965,  val_mse:0.0468,
....................................................................................................
Epoch: 6300, loss:0.0007,  mae:0.0205,  mse:0.0007,  val_loss:0.0477,  val_mae:0.1991,  val_mse:0.0477,
....................................................................................................
Epoch: 6400, loss:0.0007,  mae:0.0206,  mse:0.0007,  val_loss:0.0468,  val_mae:0.1965,  val_mse:0.0468,
....................................................................................................
Epoch: 6500, loss:0.0007,  mae:0.0206,  mse:0.0007,  val_loss:0.0468,  val_mae:0.1966,  val_mse:0.0468,
....................................................................................................
Epoch: 6600, loss:0.0007,  mae:0.0205,  mse:0.0007,  val_loss:0.0478,  val_mae:0.1991,  val_mse:0.0478,
....................................................................................................
Epoch: 6700, loss:0.0007,  mae:0.0205,  mse:0.0007,  val_loss:0.0469,  val_mae:0.1964,  val_mse:0.0469,
....................................................................................................
Epoch: 6800, loss:0.0007,  mae:0.0205,  mse:0.0007,  val_loss:0.0467,  val_mae:0.1962,  val_mse:0.0467,
....................................................................................................
Epoch: 6900, loss:0.0007,  mae:0.0205,  mse:0.0007,  val_loss:0.0476,  val_mae:0.1988,  val_mse:0.0476,
....................................................................................................
```

In [35]:
```
plotter = tfdocs.plots.HistoryPlotter(smoothing_std=2)
```

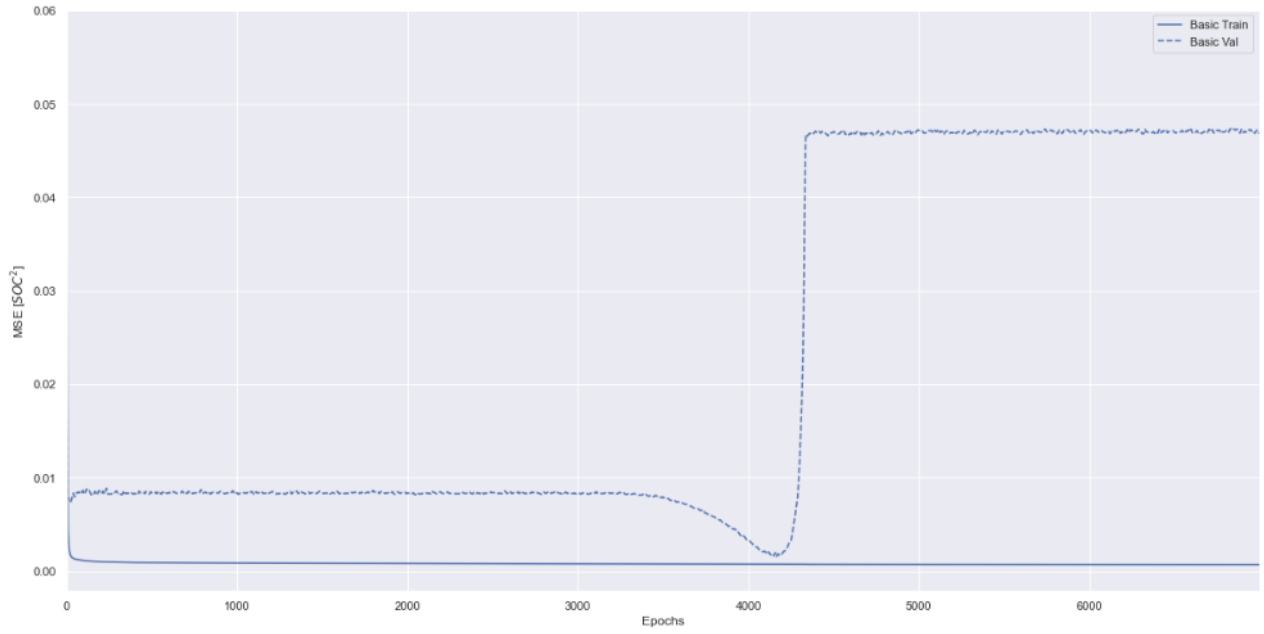
In [36]:
```
plt.figure(figsize=(20, 10))
plotter.plot({'Basic': history}, metric = "mae")
plt.ylabel('MAE [SOC]')
```

Out[36]:
```
Text(0, 0.5, 'MAE [SOC]')
```



In [44]:
```
plt.figure(figsize=(20, 10))
plotter.plot({'Basic': history}, metric = "mse")
plt.ylabel('MSE [$SOC^{2}$]')
```

Out[44]:
```
Text(0, 0.5, 'MSE [$SOC^{2}$]')
```



Validation of Model

This section uses the remaining drive cycle data sets to test the actual error of the model when used on data that is completely new to the model.

In [59]:
```
dfs = [
    pd.read_csv(os.path.join(dp, f)) for
    dp, dn, filenames in os.walk('./csv-data')
    for f in filenames if (os.path.splitext(f)[1] == '.csv')
]

df3 = pd.concat(dfs, axis=0, ignore_index=True)
del dfs
```

In [60]:
```
df3.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1126083 entries, 0 to 1126082
Data columns (total 8 columns):
 #   Column        Non-Null Count    Dtype
---  ------        --------------    -----
 0   Voltage       1126083 non-null  float64
 1   Current       1126083 non-null  float64
 2   Ah            1126083 non-null  float64
 3   Wh            1126083 non-null  float64
 4   Power         1126083 non-null  float64
 5   Battery Temp  1126083 non-null  float64
 6   Time          1126083 non-null  float64
 7   Chamber Temp  1126083 non-null  int64
dtypes: float64(7), int64(1)
memory usage: 68.7 MB
```

In [62]:
```
df3['Capacity'] = 1 - (-df3['Ah'] / 2.784)
```

In [63]:
```
X = df3.copy()
y = X['Capacity']

X = X.drop(columns=['Time', 'Wh', 'Ah', 'Capacity'])
```

In [64]:
```
mean_absolute_error(model.predict(X), y)
```

Out[64]:
```
0.06964233280510271
```

Appendix B – Gantt Chart

Daniel Woodhall

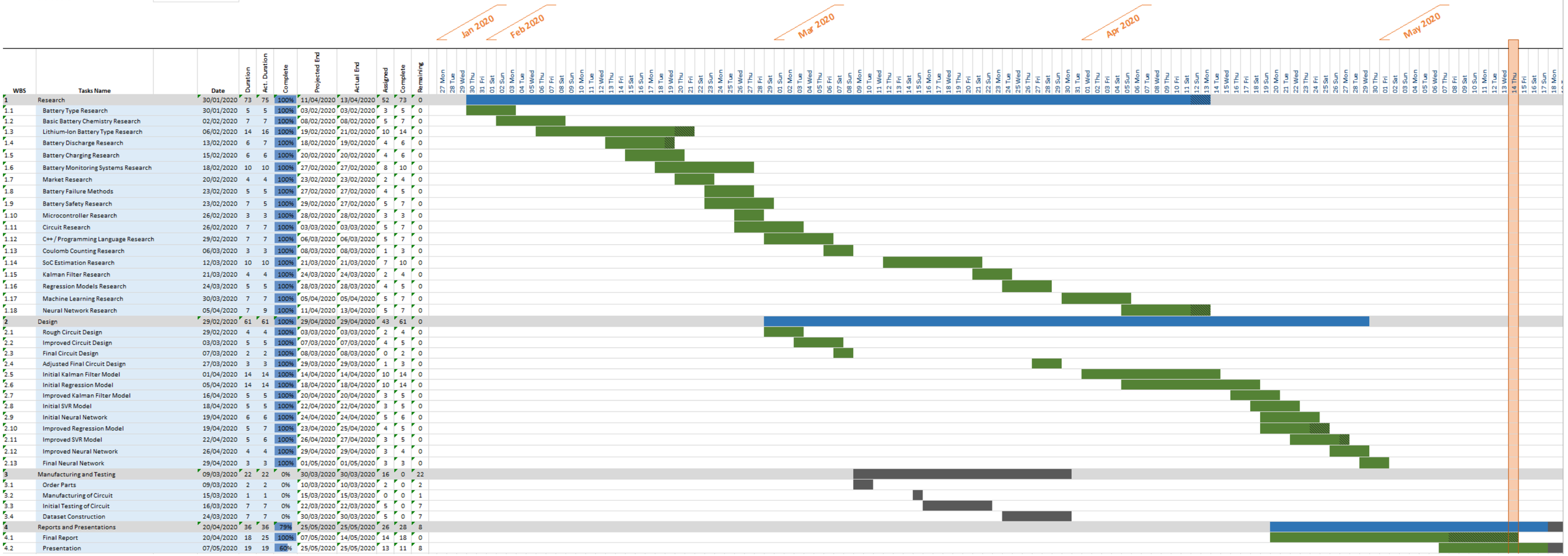
Battery Monitoring System Gantt Chart

Today's Date: 14 May 2020, Thursday

Week Starts from: Monday

Start Date: 29/01/2020

WBS	Tasks Name	Date	Duration	Act. Duration	Complete	Projected End	Actual End	Assigned	Complete	Remaining
1	Research	30/01/2020	73	75	100%	11/04/2020	13/04/2020	52	73	0
1.1	Battery Type Research	30/01/2020	5	5	100%	03/02/2020	03/02/2020	3	5	0
1.2	Basic Battery Chemistry Research	02/02/2020	7	7	100%	08/02/2020	08/02/2020	5	7	0
1.3	Lithium-Ion Battery Type Research	06/02/2020	14	16	100%	19/02/2020	21/02/2020	10	14	0
1.4	Battery Discharge Research	13/02/2020	6	7	100%	18/02/2020	19/02/2020	4	6	0
1.5	Battery Charging Research	15/02/2020	6	6	100%	20/02/2020	20/02/2020	4	6	0
1.6	Battery Monitoring Systems Research	18/02/2020	10	10	100%	27/02/2020	27/02/2020	8	10	0
1.7	Market Research	20/02/2020	4	4	100%	23/02/2020	23/02/2020	2	4	0
1.8	Battery Failure Methods	23/02/2020	5	5	100%	27/02/2020	27/02/2020	4	5	0
1.9	Battery Safety Research	23/02/2020	7	5	100%	29/02/2020	27/02/2020	5	7	0
1.10	Microcontroller Research	26/02/2020	3	3	100%	28/02/2020	28/02/2020	3	3	0
1.11	Circuit Research	26/02/2020	7	7	100%	03/03/2020	03/03/2020	5	7	0
1.12	C++ / Programming Language Research	29/02/2020	7	7	100%	06/03/2020	06/03/2020	5	7	0
1.13	Coulomb Counting Research	06/03/2020	3	3	100%	08/03/2020	08/03/2020	1	3	0
1.14	SoC Estimation Research	12/03/2020	10	10	100%	21/03/2020	21/03/2020	7	10	0
1.15	Kalman Filter Research	21/03/2020	4	4	100%	24/03/2020	24/03/2020	2	4	0
1.16	Regression Models Research	24/03/2020	5	5	100%	28/03/2020	28/03/2020	4	5	0
1.17	Machine Learning Research	30/03/2020	7	7	100%	05/04/2020	05/04/2020	5	7	0
1.18	Neural Network Research	05/04/2020	7	9	100%	11/04/2020	13/04/2020	5	7	0
2	Design	29/02/2020	61	61	100%	29/04/2020	29/04/2020	43	61	0
2.1	Rough Circuit Design	29/02/2020	4	4	100%	03/03/2020	03/03/2020	2	4	0
2.2	Improved Circuit Design	03/03/2020	5	5	100%	07/03/2020	07/03/2020	4	5	0
2.3	Final Circuit Design	07/03/2020	2	2	100%	08/03/2020	08/03/2020	0	2	0
2.4	Adjusted Final Circuit Design	27/03/2020	3	3	100%	29/03/2020	29/03/2020	1	3	0
2.5	Initial Kalman Filter Model	01/04/2020	14	14	100%	14/04/2020	14/04/2020	10	14	0
2.6	Initial Regression Model	05/04/2020	14	14	100%	18/04/2020	18/04/2020	10	14	0
2.7	Improved Kalman Filter Model	16/04/2020	5	5	100%	20/04/2020	20/04/2020	3	5	0
2.8	Initial SVR Model	18/04/2020	5	5	100%	22/04/2020	22/04/2020	3	5	0
2.9	Initial Neural Network	19/04/2020	6	6	100%	24/04/2020	24/04/2020	5	6	0
2.10	Improved Regression Model	19/04/2020	5	7	100%	23/04/2020	25/04/2020	4	5	0
2.11	Improved SVR Model	22/04/2020	5	6	100%	26/04/2020	27/04/2020	3	5	0
2.12	Improved Neural Network	26/04/2020	4	4	100%	29/04/2020	29/04/2020	3	4	0
2.13	Final Neural Network	29/04/2020	3	3	100%	01/05/2020	01/05/2020	3	3	0
3	Manufacturing and Testing	09/03/2020	22	22	0%	30/03/2020	30/03/2020	16	0	22
3.1	Order Parts	09/03/2020	2	2	0%	10/03/2020	10/03/2020	2	0	2
3.2	Manufacturing of Circuit	15/03/2020	1	1	0%	15/03/2020	15/03/2020	0	0	1
3.3	Initial Testing of Circuit	16/03/2020	7	7	0%	22/03/2020	22/03/2020	5	0	7
3.4	Dataset Construction	24/03/2020	7	7	0%	30/03/2020	30/03/2020	5	0	7
4	Reports and Presentations	20/04/2020	36	36	79%	25/05/2020	25/05/2020	26	28	8
4.1	Final Report	20/04/2020	18	25	100%	07/05/2020	14/05/2020	14	18	0
4.2	Presentation	07/05/2020	19	19	60%	25/05/2020	25/05/2020	13	11	8



Appendix C – Bill of Materials

Product	Price per unit	URL Link
Arduino UNO Rev3	£17.00	https://uk.rs-online.com/web/p/processor-microcontroller-development-kits/7697409/
Arcol HS50 Series Aluminium Housed Axial Wire Wound Panel Mount Resistor, 10KΩ, 50W Rating	£2.37	https://uk.rs-online.com/web/p/panel-mount-fixed-resistors/0159641/
RS PRO 26.1kΩ Resistor	£1.34	https://uk.rs-online.com/web/p/fixed-resistors/7176040/
KEMET 100nF Multilayer Ceramic Capacitor	£0.20	https://uk.rs-online.com/web/p/mlccs-multilayer-ceramic-capacitors/5381310/
Temperature Sensor - TMP36 (x2)	£1.44	https://www.robotshop.com/uk/temperature-sensor-tmp36.html?gclid=CjwKCAjwte71BRBCEiwAU_V9h1-S20v2-rLTG3qkhp_sjtoLOyNEGao5S2Y25DxXNURoDu3mPe-oZxoCDfsQAvD_BwE
ACS712ELCTR-50A-T Allegro Microsystems, Linear Hall Effect Sensor	£3.63	https://uk.rs-online.com/web/p/hall-effect-sensor-ics/6807135/
Coulomb Counter Breakout, LTC4150 – BOB-12052	£10.69	https://cpc.farnell.com/sparkfun-electronics/bob-12052/coulomb-counter-breakout-ltc4150/dp/MK00755?mckv=sXvttb0LR_dc|pcrid|224646539664|kword||match||plid||slid||product|MK00755|pgrid|49734053271|ptaid|pla-836865197975|&CMP=KNC-GUK-CPC-SHOPPING&gclid=CjwKCAjwte71BRBCEiwAU_V9hx2GwQyJ3FWE6aZJ-XIZ_PLs584VdzsBaN6P1qwzBNB_mY-a0ijbaxoCXq8QAvD_BwE