A PROJECT REPORT ON

Prediction of Remaining Useful Life of Lithium-ion Batteries Using ANN Model and its Comparative Study with CNN, SVM, and Linear Regression Models

Miss. Samina Attari (191081007)

Miss. Khushi Barjatia (191081009)

Miss. Mansi Kadam (191081035)

Mr. Dhruv Shah (201080901)

Mr. Neel Dandiwala (201080906)

Under the guidance of

Prof. Sandeep Udmale

DEPARTMENT OF INFORMATION TECHNOLOGY,

VEERMATA JIJABAI TECHNOLOGICAL INSTITUTE,

H.R MAHAJAN ROAD, MATUNGA, MUMBAI,

MAHARASHTRA- 400019

(2021-22)

Certificate

This is to certify that Miss Samina Attari, Miss Khushi Barjatia, Miss Mansi Kadam, Mr. Dhruv Shah, and Mr. Neel Dandiwala students of B.Tech (Information Technology), VJTI, Matunga, Mumbai - 400019 have successfully completed the project on “Prediction of Remaining Useful Life of Lithium-ion Batteries Using ANN Model and its Comparative Study with CNN, SVM, and Linear Regression Models” under the guidance of Prof Sandeep Udmale as a part of Machine Learning term work.

Prof. Sandeep Udmale

Computer Engineering, VJTI,

Mumbai - 19.

Statement by Candidate

We wish to state that the work embodied in this special topic titled “Prediction of Remaining Useful Life of Lithium-ion Batteries Using ANN Model and its Comparative Study with CNN, SVM, and Linear Regression Models” forms my own contribution to the work carried out under the guidance of Prof. Sandeep Udmale at the ‘Veermata Jijabai Technological Institute’, Matunga, Mumbai – 19. This work has not been submitted for any other Degree or Diploma of any University / Institute. Wherever references have been made to previous works of others, it has been clearly indicated.

Miss Samina Attari (191081007)

Miss Khushi Barjatia (19108100)

Miss Mansi Kadam (191081035)

Master Dhruv Shah (201080901)

Master Neel Dandiwala (201080906)

Abstract

Accurate equipment remaining useful life prediction is critical to effective condition-based maintenance for improving reliability and reducing overall maintenance cost by noticing the batteries' life periodically. An artificial neural network (ANN) based method is developed for achieving more accurate remaining useful life prediction of Lithium-Ion batteries subject to condition monitoring. The ANN model takes the capacity attribute as a target against multiple measurement values as the inputs, and the life expectancy as the output. The proposed method is validated with the data provided by Prognostics Center of Excellence at Ames Research Center, NASA. A comparative study is performed between the proposed ANN method and other methods, and the results demonstrate the advantage of the proposed method in achieving more accurate remaining useful life prediction.

Index

Problem Statement

In modern times, every device runs on electricity. Batteries are fulfilling electricity needs for portable devices. In all the latest portable electronic devices, lithium-ion batteries (LIBs) are the primary source of power: the reason being their high charge storage density. The primary issue with the LIB is the lack of information on its remaining useful life (RUL). Thus, we try to analyze and predict the RUL of LIB. We present a novel statistical approach for extracting various critical points from the discharge cycle by inspecting the LIB dataset. The inspection of LIB assisted in finding the pattern and relationship with the number of cycles. Thus, two feature sets are proposed in this paper based on critical points. The feature set is examined with various machine learning regression models using the LIB dataset. Also, the comparative analysis is performed for the proposed merging parameter to find the decent performance. Index Terms—Critical point, Lithium-ion battery (LIB), Regression, Remaining useful life (RUL).

Lithium-ion battery is introduced recently as a key solution for energy storage problems both in stationery and mobility applications. However, one main limitation of this technology is the aging, i.e., the degradation of storage capacity. This degradation happens in every condition, whether the battery is used or not, but in different proportions depending on the usage and external conditions. Due to the complexity of aging phenomena to characterize, lifetime modeling of Li-ion cells has attracted the attention of researchers in recent years. This paper develops cycling lifetime prediction models, for two different commercially available Li-ion cells, by using artificial neural networks. First, accelerated cycling tests are performed under different testing conditions, including different temperatures, state of charges, depth of discharges, and discharge current rates. Then, the test data is used to train a feedforward neural network that can predict one-step ahead state of health of the cells that are cycled under different conditions. Thereafter, a

sensitivity analysis method is used to investigate the dependence of the state of health of the cells to each input parameter by calculating the partial derivative of the neural network model output with regard to each input. Finally, the sensitivity profile over the whole range of the inputs is provided and discussed.

The project determines the RUL of the Li-ion batteries using the following approaches:

Main approach:

- ANN Model

For Comparative Analysis:

- CNN Model
- Linear Regression Model
- Support Vector Machine (SVM) Model

Dataset Description

The dataset is from NASA Ames Prognostics Center of Excellence. To simulate the dynamic operating conditions in real applications, 18650 LiCoO2 batteries (2.1 Ah) were cycled under a series of random currents rather than the constant discharge currents. Each loading period lasted for 5 minutes. A 2 A charging and discharging test was performed after every 1500 periods (about 5 days) to measure the battery capacity. For our study, we used the data from B0005, B0006, B0007, and B0018 batteries. The failure thresholds for those batteries are considered as the capacities at the end of the test.

Dataset Description:

Number Of Battery Samples: 4

The dataset has 636 records across 7 variables.

Ambient Temperature considered for collecting data: 24°C

Data Structure:

cycle: top level structure array containing the charge, discharge and impedance operations

type: operation type, can be charge, discharge or impedance

ambient_temperature: ambient temperature (degree C)

time: the date and time of the start of the cycle, in MATLAB date vector format

data: data structure containing the measurements

for charge the fields are:

Voltage_measured:	Battery terminal voltage (Volts)
Current_measured:	Battery output current (Amps)
Temperature_measured:	Battery temperature (degree C)
Current_charge:	Current measured at charger (Amps)
Voltage_charge:	Voltage measured at charger (Volts)

Time: Time vector for the cycle (secs)

for discharge the fields are:

Voltage_measured: Battery terminal voltage (Volts)

Current_measured: Battery output current (Amps)

Temperature_measured: Battery temperature (degree C)

Current_charge: Current measured at load (Amps)

Voltage_charge: Voltage measured at load (Volts)

Time: Time vector for the cycle (secs)

Capacity: Battery capacity (Ahr) for discharge till 2.7V

for impedance the fields are:

Sense_current: Current in sense branch (Amps)

Battery_current: Current in battery branch (Amps)

Current_ratio: Ratio of the above currents

Battery_impedance: Battery impedance (Ohms) computed from data

Rectified_impedance: Calibrated and smoothed battery impedance

Re: Estimated electrolyte resistance (Ohms)

Rct: Estimated charge transfer resistance (Ohms)

	A	B	C	D	E	F	G
1	Cycle	Time Measured (Sec)	Voltage Measured (V)	Current Measured (A)	Temperature Measured (degreee C)	Capacity (Ah)	SampleId
2	0	3690.234	3.277169977	-0.006528351	34.23085284	1.856487421	B0005
3	0	3690.234	2.475767757	-2.009435892	39.16298653	2.035337591	B0006
4	0	3690.234	3.062112709	-0.001433299	37.33847849	1.891052295	B0007
5	0	3434.891	3.053230339	-0.002433415	37.2056713	1.855004521	B0018
6	1	3672.344	3.300244887	-0.000447553	34.39213659	1.84632725	B0005
7	1	3672.344	2.351525512	-2.010374856	39.24620268	2.025140246	B0006
8	1	3672.344	3.079226238	-0.003230385	37.1617386	1.880637028	B0007
9	1	3425.485	3.08820017	-0.000910648	37.15547522	1.843195532	B0018
10	2	3651.641	3.32745101	0.001026019	34.23277872	1.835349194	B0005
11	2	3651.641	2.440479715	-2.008558887	38.99920246	2.013326371	B0006
12	2	3651.641	3.063265547	-0.003157783	37.50810004	1.880662672	B0007
13	2	3410.375	3.076857654	-0.000684881	36.68085193	1.839601842	B0018
14	3	3631.563	3.314181859	-0.002233622	34.41344995	1.835262528	B0005
15	3	3631.563	2.479155854	-2.009290448	38.84362777	2.013284666	B0006
16	3	3631.563	3.039420908	-0.003421518	37.85540509	1.880770901	B0007

Feature Analysis Of Dataset

NASA battery B0005 dataset:

In this project, a dataset of commercial Type 1850 Lithium-ion batteries from the Prognostics Center of Excellence (PCoE), NASA is used.

For battery #5 the capacity rating is 2.0Ahr. In the experiment, the battery repeatedly underwent charge, discharge, and impedance measures at room temperature (24°C). Specifically:

Charging: Charge the battery with a constant electric current of 1.5A until the voltage reaches 4.2V. Then, charge it with constant voltage until the charging current decreases to 20mA. Stop charging.

Discharging: Discharge with 2A current until the voltage declined to around 2.7V

The experiment terminated when the capacity of the battery declined to about 70% of the capacity ratings (the given failure threshold)

Import the dataset:

```
import pandas as pd
```

```
data = pd.read_csv('Input n Capacity.csv')
```

```
#Format the data
data = data.drop(labels=['SampleId'], axis=1)
dataset = data[~data.isin(['?'])]
dataset = dataset.dropna(axis=0)
dataset = dataset.apply(pd.to_numeric)
```

The first data source is the charge time-series data which consists of **Cycle, Time Measured, Voltage Measured, Current Measured, Temperature Measured, and Capacity**. Below, we display the records in the dataset. A summary of the whole dataset is also provided.

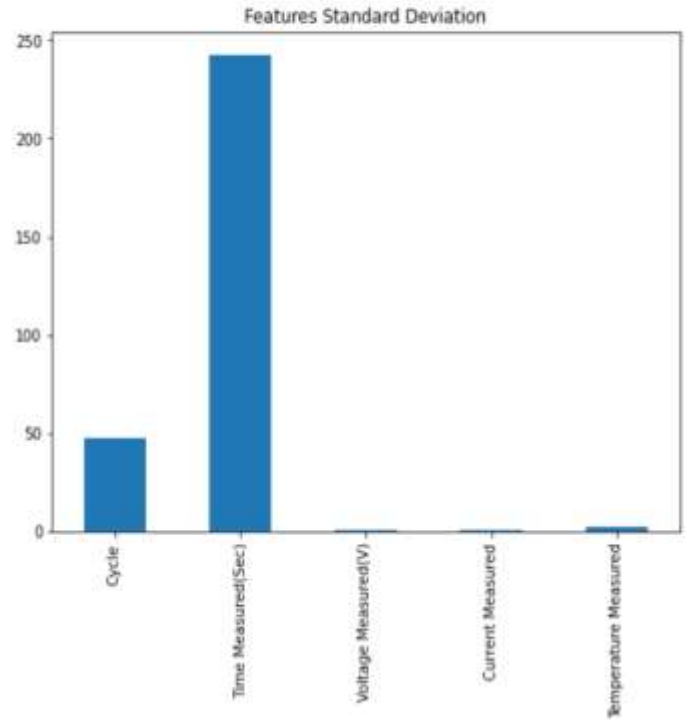
```
dataset.describe()
```

	Cycle	Time Measured(Sec)	Voltage Measured(V)	Current Measured	Temperature Measured	Capacity(Ah)
count	636.000000	636.000000	636.000000	636.000000	636.000000	636.000000
mean	79.764151	3116.977701	3.297086	-0.171153	36.318064	1.581652
std	47.137103	242.197224	0.382406	0.556974	2.090171	0.198765
min	0.000000	2742.843000	1.813269	-2.012015	32.113473	1.153818
25%	39.000000	2891.996250	3.260587	-0.003576	34.639503	1.421123
50%	79.000000	3084.281000	3.397571	-0.001903	35.808964	1.559695
75%	119.000000	3311.828000	3.529257	-0.000338	38.447301	1.763486
max	167.000000	3690.234000	3.697170	0.009113	41.049942	2.035338

The Graph below describes the standard deviation for all the features:

```
features=['Cycle','Time Measured(Sec)','Voltage Measured(V)','Current Measured','Temperature Measured']
dataset[features].std().plot(kind='bar', figsize=(8,6), title="Features Standard Deviation")
```

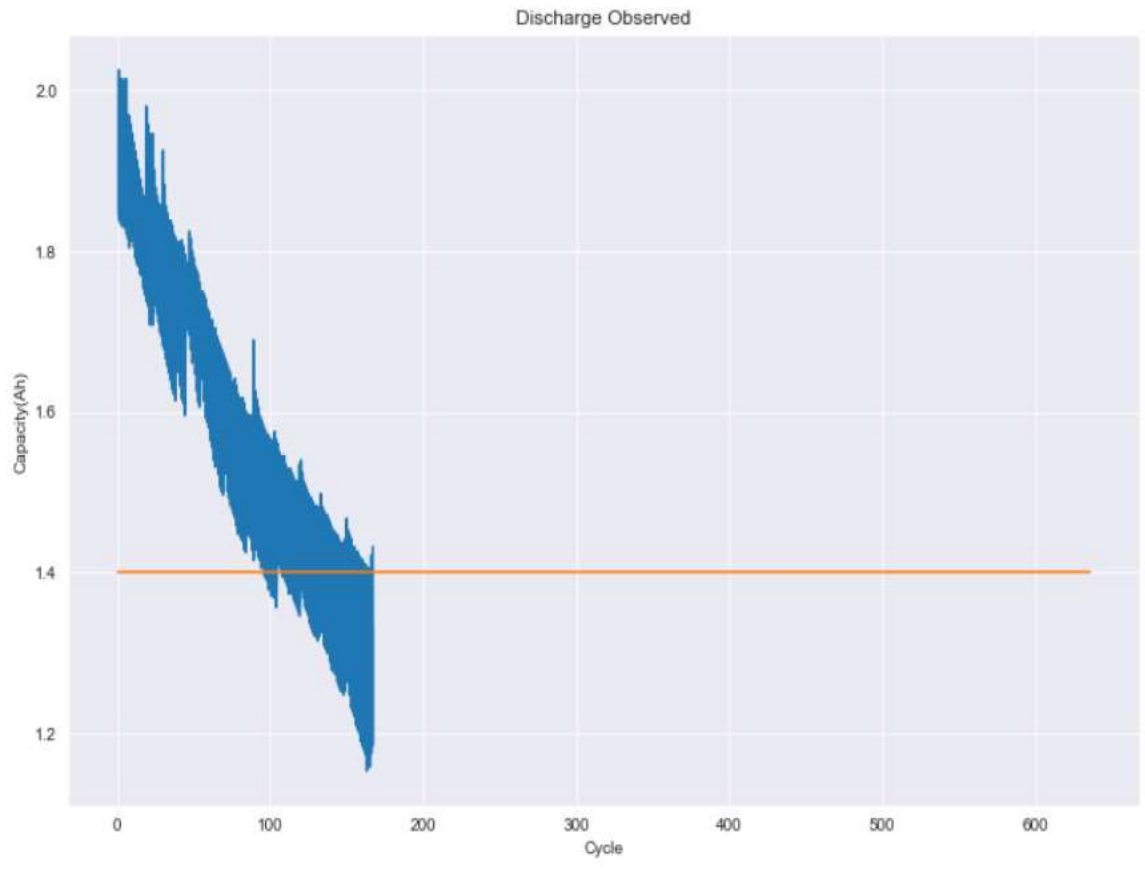
```
<AxesSubplot:title={'center':'Features Standard Deviation'}>
```



The figure below shows capacity changes of the observed samples over the charge-discharge cycles.

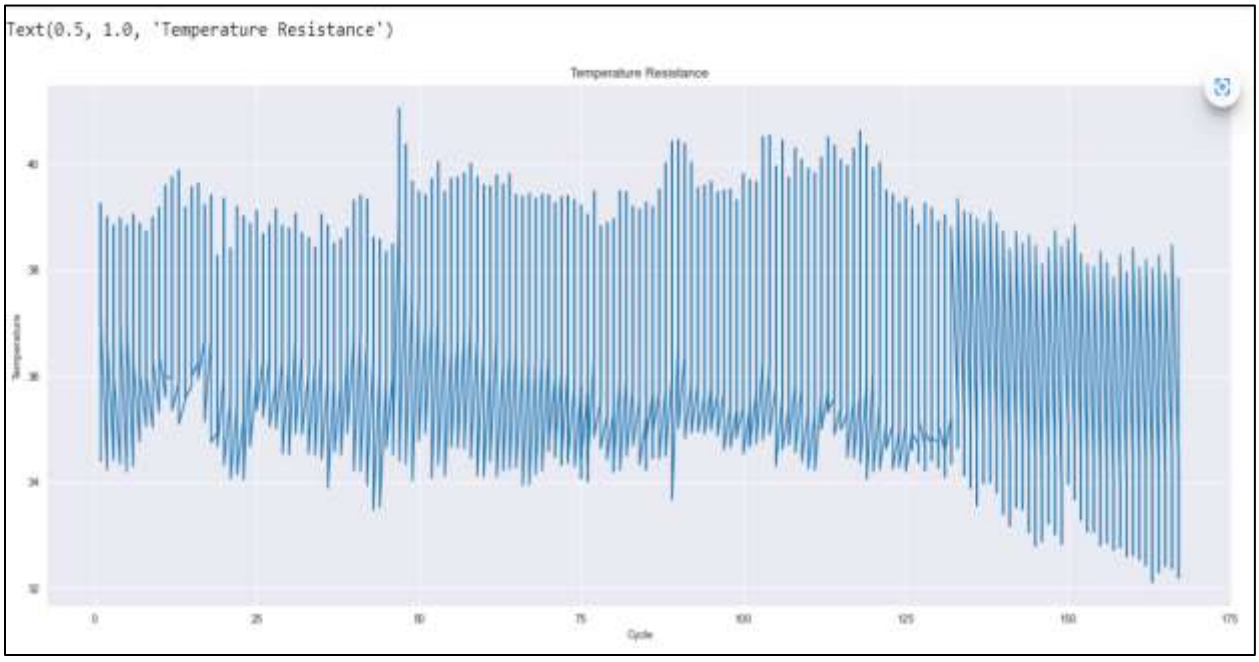
```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
plot_df = dataset.loc[(dataset['Cycle']>=1),['Cycle','Capacity(Ah)']]
sns.set_style("darkgrid")
plt.figure(figsize=(12, 8))
plt.plot(plot_df['Cycle'], plot_df['Capacity(Ah)'])
#Draw threshold
plt.plot([0.,len(dataset)], [1.4, 1.4])
plt.ylabel('Capacity(Ah)')
# make x-axis ticks legible
adf = plt.gca().get_xaxis().get_major_formatter()
plt.xlabel('Cycle')
plt.title('Discharge Observed')
```

```
Text(0.5, 1.0, 'Discharge Observed')
```



Plotting discharge data for the observed samples

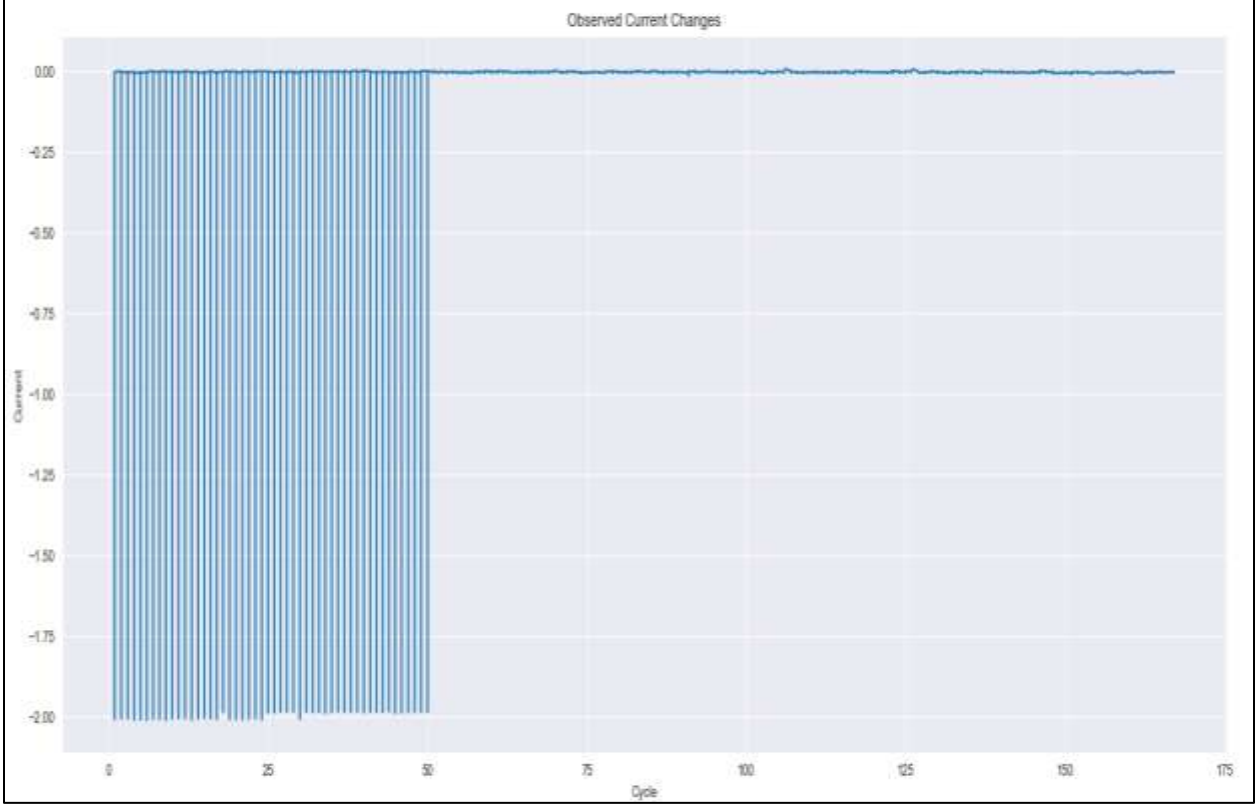
```
plot_df = dataset.loc[(dataset['Cycle']>=1),['Cycle','Temperature Measured']]
sns.set_style("darkgrid")
plt.figure(figsize=(20, 8))
plt.plot(plot_df['Cycle'], plot_df['Temperature Measured'])
plt.ylabel('Temperature')

# make x-axis ticks legible
adf = plt.gca().get_xaxis().get_major_formatter()
#adf.scaled[1.0] = '%m-%d-%Y'
plt.xlabel('Cycle')
plt.title('Temperature Resistance')
```



Plotting Current measured - Current observed over the number of cycles

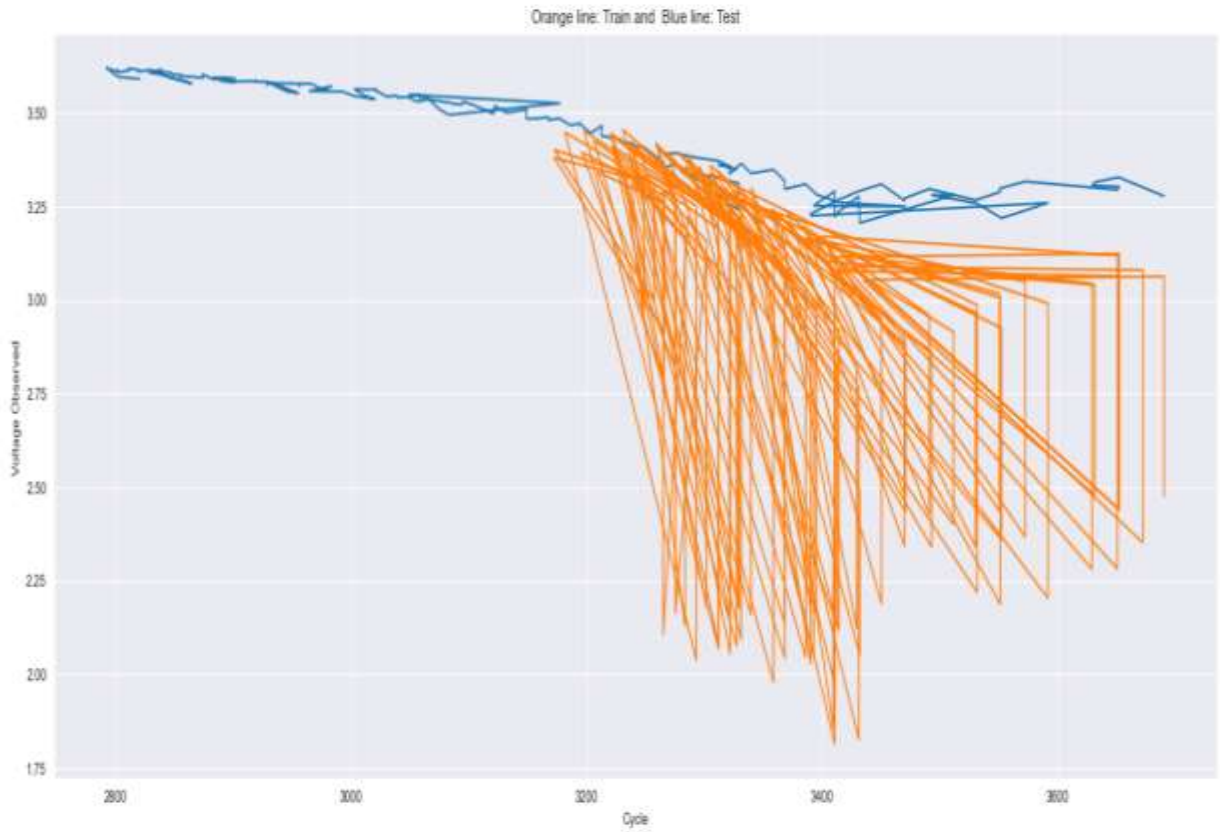
```
plot_df = dataset.loc[(dataset['Cycle']>=1),['Cycle','Current Measured']]
sns.set_style("darkgrid")
plt.figure(figsize=(20, 8))
plt.plot(plot_df['Cycle'], plot_df['Current Measured'])
plt.ylabel('Current')

# make x-axis ticks legible
adf = plt.gca().get_xaxis().get_major_formatter()
#adf.scaled[1.0] = '%m-%d-%Y'
plt.xlabel('Cycle')
plt.title('Observed Current Changes')
```

```
Text(0.5, 1.0, 'Observed Current Changes')
```




```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
test = pd.read_csv('testing_data.csv')
train = pd.read_csv('training_data.csv')
time12=train['Time Measured(Sec)'][280]
print(time12)
#test['Time Measured(Sec)']=test['Time Measured(Sec)']+time12

plot_df = test.loc[(test['Cycle']),['Time Measured(Sec)','Voltage Measured(V)']]
plot_charge=train.loc[(train['Cycle']),['Time Measured(Sec)','Voltage Measured(V)']]
sns.set_style("darkgrid")
plt.figure(figsize=(20, 8))
plt.plot(plot_df['Time Measured(Sec)'], plot_df['Voltage Measured(V)'])
plt.plot(plot_charge['Time Measured(Sec)'], plot_charge['Voltage Measured(V)'])
#Draw threshold
#plt.plot(dis['cycle'], dis['limt']) 'g'
plt.ylabel('Voltage Observed')

# make x-axis ticks legible
adf = plt.gca().get_xaxis().get_major_formatter()
#adf.scaled[1.0] = '%m-%d-%Y'
plt.xlabel('Cycle')
plt.title('Orange line: Train and  Blue line: Test')
```

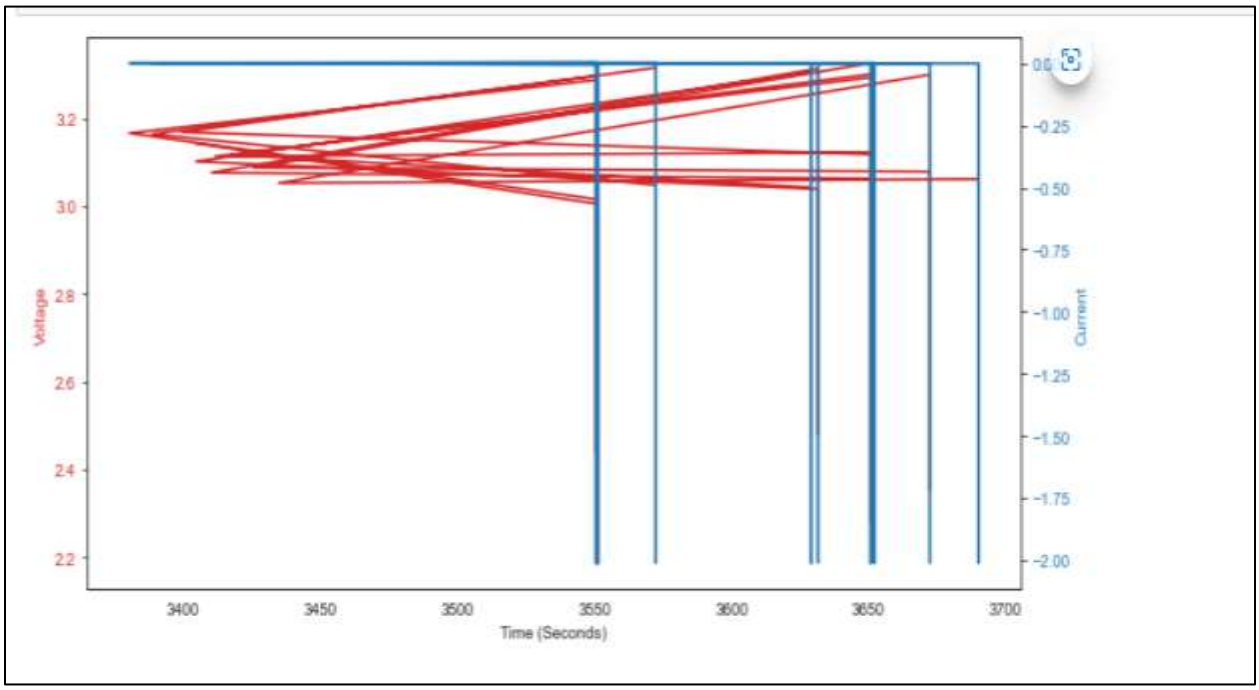
```
3056.437

Text(0.5, 1.0, 'Orange line: Train and  Blue line: Test')
```




```
# aget ordered list of top variance features:

features_top_var = dataset[features].std().sort_values(ascending=False)
features_top_var
```

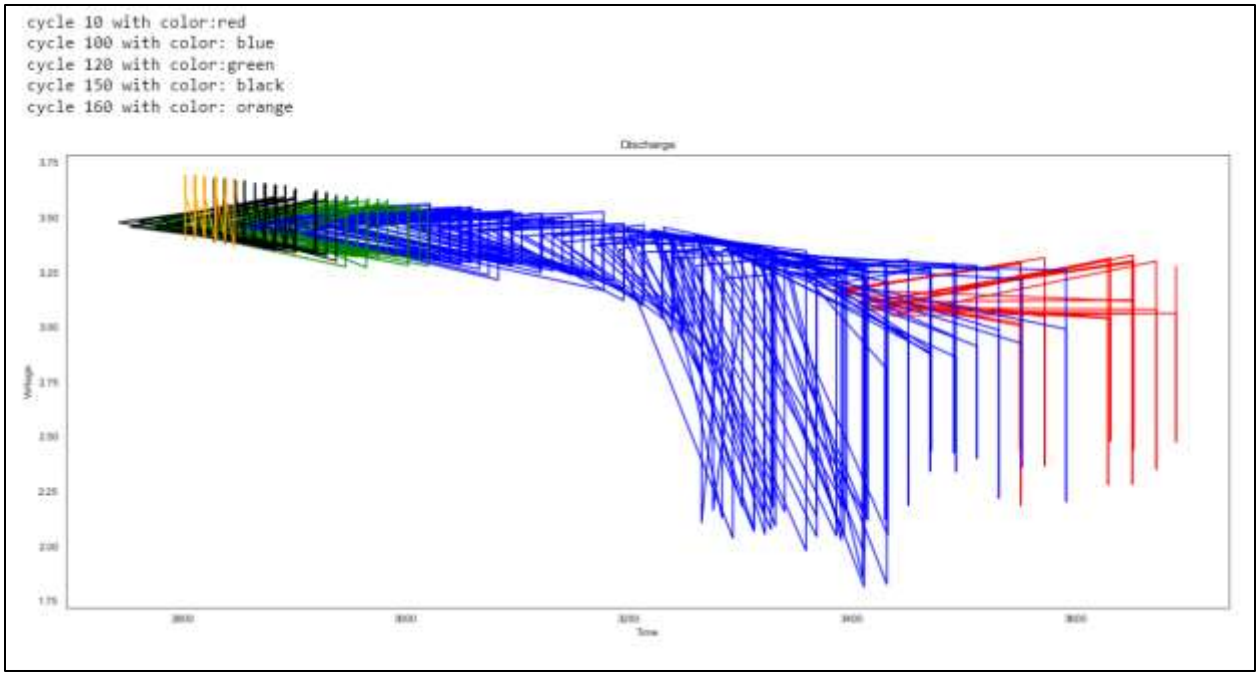
```
Time Measured(Sec)        242.197224
Cycle                      47.137103
Temperature Measured        2.090171
Current Measured            0.556974
Voltage Measured(V)         0.382406
dtype: float64
```

Figure below shows the Li-ion charge and discharge process in one cycle. The graph explains the current and voltage relation.



The figures below explain (the cycles) that after long-term, repeated charges and discharges, the lifetime of the Li-ion battery will be gradually reduced due to some irreversible reactions.

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns

plot_df = dataset.loc[(dataset['Cycle']<10),['Time Measured(Sec)','Voltage Measured(V)']]
plot_df1 = dataset.loc[(dataset['Cycle'] > 10) & (dataset['Cycle'] < 100),['Time Measured(Sec)','Voltage Measured(V)']]
plot_df2 = dataset.loc[(dataset['Cycle'] > 100) & (dataset['Cycle'] < 120),['Time Measured(Sec)','Voltage Measured(V)']]
plot_df3 = dataset.loc[(dataset['Cycle'] > 120) & (dataset['Cycle'] < 150),['Time Measured(Sec)','Voltage Measured(V)']]
plot_df4 = dataset.loc[(dataset['Cycle'] > 150) & (dataset['Cycle'] <= 160),['Time Measured(Sec)','Voltage Measured(V)']]


sns.set_style("white")
plt.figure(figsize=(20, 8))
plt.plot(plot_df['Time Measured(Sec)'], plot_df['Voltage Measured(V)'],color='red')
plt.plot(plot_df1['Time Measured(Sec)'], plot_df1['Voltage Measured(V)'],'-',color='blue')
plt.plot(plot_df2['Time Measured(Sec)'], plot_df2['Voltage Measured(V)'],color='green')
plt.plot(plot_df3['Time Measured(Sec)'], plot_df3['Voltage Measured(V)'],'-',color='black')
plt.plot(plot_df4['Time Measured(Sec)'], plot_df4['Voltage Measured(V)'],color='Orange')


plt.ylabel('Voltage')

# make x-axis ticks legible
adf = plt.gca().get_xaxis().get_major_formatter()
#adf.scaled[1.0] = '%m-%d-%Y'
plt.xlabel('Time')
plt.title('Discharge')
print(" cycle 10 with color:red\n cycle 100 with color: blue\n cycle 120 with color:green\n cycle 150 with color: black\n cycle
```



After analyzing all datasets, we find the data is heterogeneous because it is sourced from different batteries that have operated under varying temperature, current, voltage and load condition.

Motivation

The main motivation behind this project was **to learn the features of artificial neural networks and compare them with the computational neural networks for predicting the Remaining Useful Life (RUL) of Li-ion batteries.**

Li-ion batteries are widely used in consumer electronics, electric vehicles and space systems. However, a Li-ion battery has a useful life, that means with continuous charge and discharge cycles and material aging, battery performance will continue to decline until it fails to function. To predict the remaining useful life (RUL) is an effective way to indicate the health of lithium-ion batteries, which will help to improve the reliability and safety of battery-powered systems. Remaining life of a Li-ion battery is also known as battery cycle life, referring to the number of complete charge/discharge cycles that the battery can support before its capacity falls under 70% of its original capacity.

It is known that the capacity of a Li-ion battery is continuously declining after every charge and discharge cycle, and the degradation trend is very consistent. When a battery capacity drops under the failure threshold, the cell is considered to be not usable. Theoretically, it is possible to predict the remaining life of a Li-ion battery by establishing a life model of a battery. A battery life model can have many applications. Remaining useful life (RUL) is the length of time a machine is likely to operate before it requires repair or replacement. By taking RUL into account, engineers can schedule maintenance, optimize operating efficiency, and avoid unplanned downtime. For this reason, estimating RUL is a top priority in predictive maintenance programs.

- **WHY ANN MODEL?**

In equipments running on Lithium Ion batteries, condition monitoring data, such as number of discharges and charges, time related data, and many other factors are collected and processed to determine the battery's health condition; Future health condition and thus the remaining useful life (RUL) of the battery is predicted; and optimal maintenance actions can thus be scheduled based on the predicted future battery health condition, so that preventive replacements can be performed to prevent unexpected failures and minimize total maintenance costs. Accurate remaining life prediction is the critical to effective implementation of condition based maintenance pertaining to the batteries' capability and chemical combination. Existing battery health condition and RUL prediction methods can be roughly classified into model-based (or chemistry/physics-based) methods and data-driven methods. The model-based methods predict the remaining useful life using damage propagation models based on damage mechanics. If properly used, such models can greatly improve the RUL prediction accuracy. However, discharge-charge processes are typically very complex, and authentic physics based models are difficult to build for many components and systems. Data-driven methods utilize collected condition monitoring data for RUL prediction. Artificial neural networks (ANNs) have been considered to be very promising tools for battery health condition and RUL prediction due to their adaptability, nonlinearity, and arbitrary function approximation ability. Neural network methods do not assume the analytical model of the degradation propagation, but aim at modeling the degradation process, based on the collected condition monitoring data using neural networks and perform battery's health condition prediction.

In this paper, we propose an ANN-based method for achieving accurate RUL prediction of Lithium Ion batteries. The ANN model takes the capacity and multiple condition monitoring measurement values (Voltage, Current, Temperature, Cycle) at discrete timed inspection points as the inputs and the life percentage as the output. The prediction accuracy is improved mainly by careful implementation of necessary activation functions

that are relevant to the batteries' degradation in the condition monitoring data, and by utilizing the validation mechanism in the ANN training process.

- **WHY CNN MODEL?**

Lithium-ion battery is introduced recently as a key solution for energy storage problems both in stationary and mobility applications. However, one main limitation of this technology is the aging, i.e., the degradation of storage capacity. This degradation happens in every condition, whether the battery is used or not, but in different proportions depending on the usage and external conditions.

Due to the complexity of aging phenomena to characterize, lifetime modeling of Li-ion cells has attracted the attention of researchers in recent years. This paper develops cycling lifetime prediction models, for two different commercially available Li-ion cells, by using artificial neural networks. First, accelerated cycling tests are performed under different testing conditions, including different temperatures, state of charges, depth of discharges, and discharge current rates.

Then, the test data is used to train a feedforward neural network that can predict one-step ahead state of health of the cells that are cycled under different conditions. Thereafter, a sensitivity analysis method is used to investigate the dependence of the state of health of the cells to each input parameter by calculating the partial derivative of the neural network model output with regard to each input. Finally, the sensitivity profile over the whole range of the inputs is provided and discussed. Accurately predicting remaining useful life (RUL) of lithium batteries with nonlinear characters is essential for ensuring safety of applications. However, the diverse aging mechanism gives the challenge for present technologies. In this paper, a convolutional neural network (CNN) model is constructed for RUL prediction of lithium batteries. For reducing the training time of CNN model, the orthogonal method is applied for optimizing model parameters. Then, the proposal is validated by a large dataset. And the accuracy of RUL prediction exceeds 90.9 percent while root mean square

error and mean absolute error are limited to 35.1 and 13.7, respectively. The proposed method is suitable for RUL prediction of lithium batteries applied in electric vehicles and energy storage devices.

- **WHY REGRESSION MODEL?**

Regression models cannot work properly if the input data has errors (that is poor quality data). If the data preprocessing is not performed well to remove missing values or redundant data or outliers or imbalanced data distribution, the validity of the regression model suffers. Accuracy is better achieved in ANN model .While linear regression modeling approach was deficient to predict desired parameters, more accurate results were obtained with the usage of ANN.

Artificial neural networks are algorithms that can be used to perform nonlinear statistical modeling and provide a new alternative to logistic regression, the most suitable in our case for developing predictive models for RUL prediction. Neural networks offer a number of advantages, including requiring less formal statistical training, ability to implicitly detect complex nonlinear relationships between dependent and independent variables, ability to detect all possible interactions between predictor variables, and the availability of multiple training algorithms.

- **WHY SVM MODEL?**

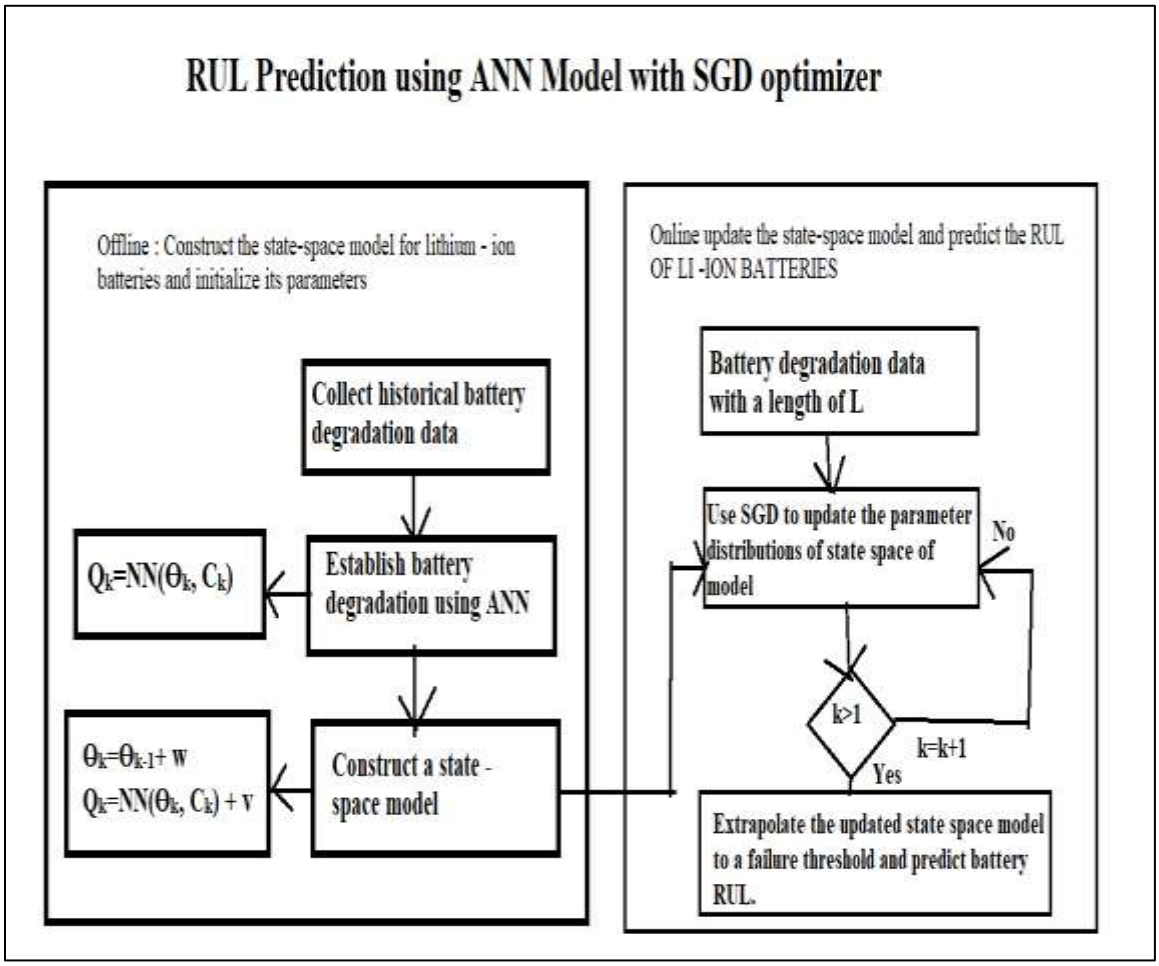
SVM has satisfactory performance in nonlinear and high-dimensional models, can deal with local minima and small sample sizes, and has a short calculation time.Satisfactory performance in non-linear and high-dimension models. Capable of dealing with the local minimum, non-linear, and small sample size problems.The global optimal solution can be obtained. High prediction accuracy.Less prediction times. Non-parametric However, it cannot express uncertainty due to its difficulty with calculating kernel and regularization parameters.

Artificial neural networks are algorithms that can be used to perform nonlinear statistical modeling and provide a new alternative to logistic regression, the most suitable in our case for developing predictive models for RUL prediction. Neural networks offer a number of advantages, including requiring less formal statistical training, ability to implicitly detect complex nonlinear relationships between dependent and independent variables, ability to detect all possible interactions between predictor variables, and the availability of multiple training algorithms.

Methodology

- **METHODOLOGY USED FOR ANN:**

As per the proposed methodology, the RUL(remaining number of charging and discharging cycles) of the Li-ion batteries are predicted using two parameters - number of cycles of battery and its known capacity. The extrapolation techniques for neural networks have been used by giving input as the number of cycles of the given battery dataset. Using this method capacities for a particular number of cycles beyond this cycle are calculated by incrementing the cycle number.



When the capacity of particular cycle reaches the threshold capacity value for the given battery then the RUL for that battery for the given cycle number is predicted as the difference between the current cycle number for which threshold capacity is reached and the input cycle number i.e. Predicted RUL = Threshold cycle number - Current input cycle number For example, if the battery is currently at cycle number 10, then we will start predicting the capacities for further cycles until the threshold capacity is reached. Let's say that the threshold capacity is reached at the 20th cycle of the battery, then at present the battery has 20 - 10 = 10 cycles remaining useful life. The real RUL is calculated as the difference of the total number of life cycles a battery has and the current cycle number at which the RUL has to be predicted. Real RUL = Total no. of cycles - Current input cycle number For example, if the total number of cycles for a battery sample is 600 and we want to calculate RUL(i.e remaining number of charging and discharging cycles) for 400 th cycle then the real RUL is 600-400= 200. The RUL of testing battery i.e B0005 is predicted from capacity and cycle number. The prediction results are based on NN methods and show the desirable properties, that is the prediction curves can converge to the real capacity curves and the RUL pdfs become narrow as the time of prediction advances. The NN method tracks the aging variation well when collecting the data after sudden changes (100 days). This is because by adjusting the model parameters using more capacity data, the NN model can effectively track the degradation trend and accordingly, achieve good prediction results. However, the capacity prediction curves obtained by the EXP model are obviously different from the real ones for all prediction times. The reason is that the degradation characteristic for this battery is relatively complex with strong dynamics. Thus, it is difficult to be tracked and predicted using the simple EXP model. The SGD method is used for optimization purposes to improve the prediction performance.

Procedure of the proposed ANN method:

The explanation of the procedure is given as follows. We start from the available historical data, which includes the numerical values and actual condition monitoring measurement values.

- **Step 1**: Each measurement data for a battery is fitted in a collective dataset. The capacity values and the fitted measurement values of various mat datasets are used to construct the ANN training set.
- **Step 2**: The ANN validation set is constructed using the collective dataset and the actual measurement values as inspection points are the Capacity values.
- **Step 3**: Train the ANN model based on the training set and the validation set using the sequential model. This Step 3 gives the trained ANN model desired obtained for RUL prediction.
- **Step 4**: At every iteration, we first fit each measurement series based on the measurement values up to the current range. The fitted measurement values as well as the target values in these two sets, are used as the inputs to the trained ANN model.
- **Step 5**: The predicted remaining useful life from the current set of values is calculated using the trained ANN model.
- **Step 6**: The RUL is calculated based on the current capacity of the battery and the measured attributes.
- **Step 7**: For every epoch, repeat Step 4 to Step 6 based on the available data, and determine the RUL prediction.

- **METHODOLOGY USED FOR CNN**

CNN can maximize the map relations between the input and the output by extracting the feature matrix. When the CNN model is trained, it can fully learn the feature matrix of input without exacting mathematical expression between the input and the output. After model training, CNN has high performance of features recognition for target prediction. Therefore, CNN is selected for RUL prediction. Compared to other neural networks, CNN benefits from three core ideas: local receptive field, weight sharing mechanism, and pooling sample. The framework of the RUL prediction by CNN is shown below.

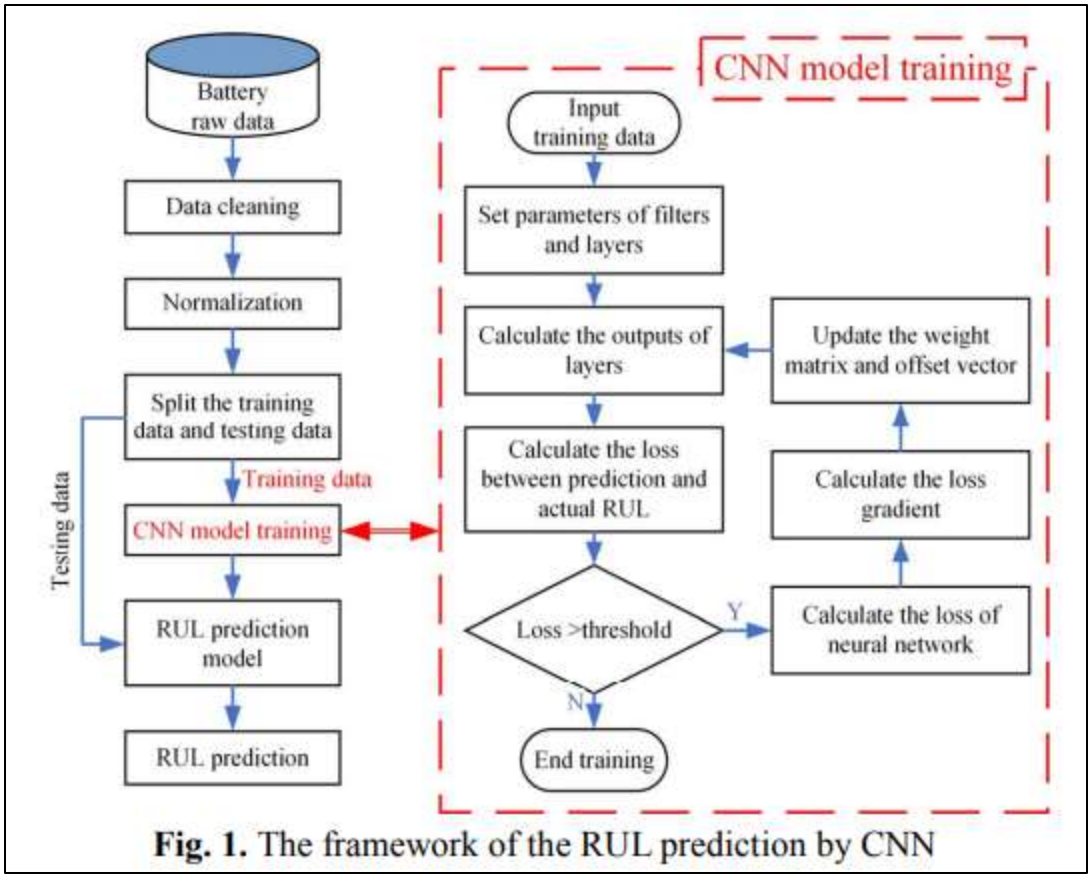


Fig. 1. The framework of the RUL prediction by CNN

Firstly, CNN uses sparse connection of local areas (local area of filter) to reduce the number of parameters in the model and the training time, and improve the performance of CNN. Then, the convolution kernel of the convolutional layer adopts the kernel parameter (weight of parameter) to finish convolution operations. The weight matrix and the offset vector are shared, which not only greatly reduces the number of network parameters, but also improves the training efficiency of CNN. Finally, a pooling sample is formed as the pooling layer that further reduces the output dimensions of the feature matrix extracted from the convolutional layer. The local feature scaling of the input feature matrix from the previous layer is performed by the mean function of the pooling layer (maximum pooling, mean pooling, etc.). So, the dimension of the feature matrix and computational burden can be further reduced.

The CNN used in this paper contains a multi-layer neural network, which consists of input layer, hidden layer, fully connected layer, and output layer. The hidden layer is composed

of many convolution layers and pooling layers. Each layer of the network consists of a two-dimensional plane containing a set of neurons. The convolutional layer is composed of various feature matrices, which can be regarded as a feature extraction layer. Assume that the original data matrix of the input layer is X, and the i th layer is the feature map (character matrix) corresponding to the convolutional layer FMi. The convolution process can be defined as:

$$\boldsymbol{FM}_i = f(\boldsymbol{FM}_{i-1} \oplus \boldsymbol{w}_i + \boldsymbol{b}_i) \qquad (1)$$

where, FMi the feature matrix corresponding to the i th convolutional layer; FMi-1 the feature matrix corresponding to the (i-1) th convolutional layer; wi the weight matrix corresponding to the i th convolutional layer; the operator of convolution; f the activate function. The operation of convolution is shown as (2).

$$\boldsymbol{y}^{l(i,j)} = \boldsymbol{CK}_i^l \oplus \boldsymbol{X}^{l(r_j)} = \sum_{j^*=0}^{C-1} \boldsymbol{CK}_i^{l(j^*)} \boldsymbol{X}^{l(j+j^*)} \qquad (2)$$

where, CKl(j *) i the j *the weight of i th kernel at l th convolution layer; Xl(rj) the j th convolution area of l th convolution layer; Xl(j+j*) the character value of the convolution area of the l th convolution layer; C the kernel width. The pooling layer connects adjacent convolutional layers and consists of multiple feature maps. Thus, the convolutional layer can be considered as the input layer of the pooling layer. Similarly, the convolution kernel is used to perform sliding calculation on the convolution layer feature map to obtain the feature map of the layer. The pooling layer can scale the feature data of the convolution layer, thereby reducing the feature data dimension and reducing the computational complexity. At the same time, the pooling layer can also perform secondary feature extraction on its feature data. Commonly used pooling calculation methods include max pooling and average pooling. In this paper, the method of max pooling is adopted because it can obtain the eigenvalues that have no positional relationship with the input feature matrix, and enhance the robustness, which is defined as follows:

$$p^{l(i,j)} = \max_{[(j-i)C+1] \le t \le jC} (f^{l(i,t)}) \quad (3)$$

where, f l(i,t) the t th activation value of i th area at l th convolution layer (the character value in the local area connecting the input matrix and pooling layer); C the width of the pooling layer; pl(i, j) the output of pooling. The fully connected layer is connected after multiple convolutional and pooling layers in the CNN. Each neuron in the fully connected layer and the neurons in the upper layer are fully connected, and the characteristic matrix of the convolutional layer or the pooled layer is spread to obtain the input feature vector of the fully connected layer, after several fully connected layers and the entire volume. The output layers of the neural network are connected to obtain the final CNN output value. Each neuron between the fully connected layers also uses an activation function (ReLU, tanh function, etc.) to improve the performance of CNN. Mean absolute error (MAE) and root mean square error (RMSE) are adopted as error indexes of RUL prediction.

$$\mathrm{MAE} = \frac{1}{n}\sum_{i=1}^{n}\frac{\left|l(i) - \hat{l}(i)\right|}{l(i)} \quad (4)$$

$$\mathrm{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(l(i) - \hat{l}(i))^2} \quad (5)$$

Optimization of Model Parameters :

The CNN model contains various parameters which have influence on the model precision in the training process. Due to the limitation of computational burden and training time, model parameters should be quickly and properly adjusted to get the optimized model. Orthogonal experiment is one of the effective methods in searching for main factors without conducting full parameter experiments. As expected, the main parameters of the CNN model can be settled. When the structure of CNN is defined, the most influenced parameters include the segment length, time distance, filter size, filter number, pooling size, dropout rate and activate function. For the data sheet for RUL prediction, the orthogonal table for three levels of seven factors .

$$A_k = \frac{1}{k}\sum_{i}^{n} S_i$$
$$R = \max\{A_k\} - \min\{A_k\} \quad (6)$$

where Sn (n = 1~6) and Ak (k = 1, 2, 3) are the sum value and mean value of prediction accuracy corresponding to the k level of factor, respectively; R is the range of Ak. The range represents the influence ratio of each factor among all factors. So, the main factors of CNN model are dropout rate, activate function and filter size. In the training process, the main factors should be adjusted firstly, then the secondary factors can be adapted. From the results in Table 1, the segment length, time distance, filter size, filter number, pooling size, dropout rate and activate function should be selected as 600, 100, 6, 50, 3, 0 and ReLU, respectively.

Procedure of the proposed CNN model:

- **Step 1 :** Reading the dataset , the raw battery data and filtering it.
- **Step 2 :** Data cleaning is the process of fixing or removing incorrect, corrupted, incorrectly formatted, duplicate, or incomplete data within a dataset. When combining multiple data sources, there are many opportunities for data to be duplicated or mislabeled. Cleaning of data for accurate results.
- **Step 3:** Normalization of data to give equal weights/importance to each variable so that no single variable steers model performance in one direction just because they are bigger numbers.
- **Step 4 :** Splitting of data into testing and training datasets.
- **Step 5 :** CNN model training which includes the following :
  - Input the training data.
  - Set parameters or filters or layers.
  - Calculate output of layers.
  - Calculate the loss between prediction and actual RUL.

- If loss> threshold , then calculate the loss of the neural network , calculate the loss of gradient and then update the weight matrix and offset vector.
- If loss<threshold , end training.

❖ **Step 6 :** RUL Prediction model

❖ **Step 7:** RUL model.

- **METHODOLOGY USED FOR LINEAR REGRESSION**

Since the RUL prediction can be classified as regression problem so the evaluation can be done on the basis of the Mean Square Error (MSE) and R- Square values. MSE represents the deviation between predicted value and actual value, smaller the test MSE better is the prediction . R-Square explains the proportion of variation within predicted value (Y) explained by the independent set of features .

Procedure of the proposed Regression Model:

❖ **Step 1:** Reading the input data.

❖ **Step 2:** Calculating charge for all cycles and Calculating the charge for the first and last cycles.

❖ **Step 3:** Calculating discharge for all cycles and Calculating the discharge for the first and last cycles.

❖ **Step 4:** Splitting model into training and testing .Discharge Cycles are used for training and testing the model.

❖ **Step 5:** Feature Extraction: In the given dataset every cycle is represented by a set of arrays.Out of which Temperature, VoltageMeasured, VoltageLoad seems to best describe the cycle . These values are measured at different time points which are represented in the Time array . Rather than using an entire array for training we can extract **critical time points for each of the features** and train the model on these **critical time points.**Only using these **critical points** will reduce the training time and reduce the noise in data.

- **Step 6:** Calculating Critical Points for a given cycle.
  - **TEMPERATURE_MEASURED :** Time at highest temperature
  - **VOLTAGE_MEASURED :** Time at lowest Voltage
  - **VOLTAGE_LOAD :** First time it drops below 1 volt after 1500 time
- **Step 7:**Training the regression model and calculating its accuracy.
- **Step 8:**Predicting the value of RUL using a regression model.

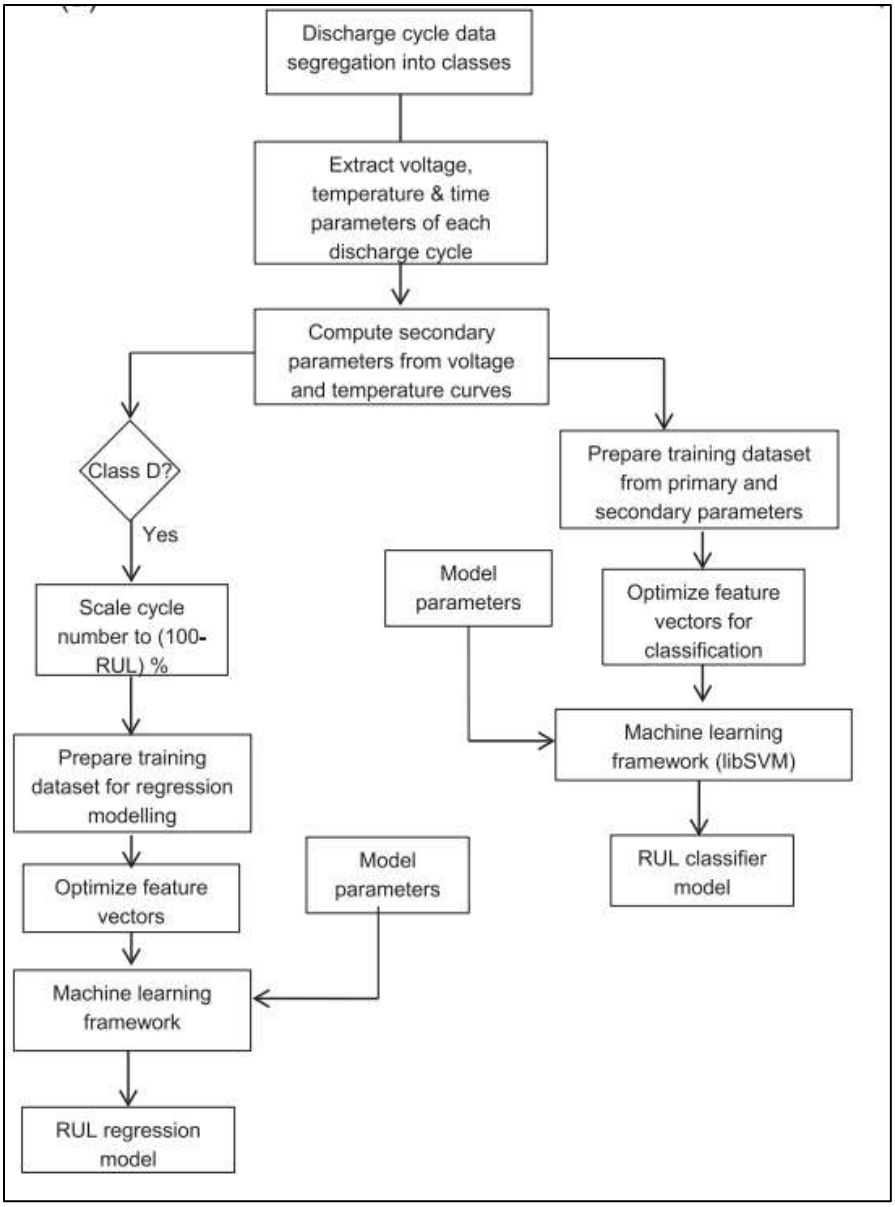
- **METHODOLOGY USED FOR SVM**

The estimation of the remaining useful life (RUL) of lithium-ion (Li-ion) batteries is important for intelligent battery management systems (BMS). Data mining technology is becoming increasingly mature, and the RUL estimation of Li-ion batteries based on data-driven prognostics is more accurate with the arrival of the era of big data. However, the support vector machine (SVM), which is applied to predict the RUL of Li-ion batteries, uses the traditional single-radial basis kernel function.

The objective of this model is to design an efficient two-stage RUL estimation system that can predict the remaining life of a battery at any stage of its life.

Below diagram provides the block diagram of this methodology. The motivations behind design of such a two-stage system are twofold:

a. For on-board scenarios, accurate RUL estimation is required only when the battery is close to end of life as against fresh.
b. Addition of a classification step before the regression step eliminates the need to perform regression across the complete battery life cycle data. Hence due to introduction of this step, heavy computations can be eliminated. This section provides a detailed description of the proposed methodology used in development of classification and regression models.



Procedure of the proposed SVM Model:

- **Step 1: Data processing and feature extraction** :- In this work, data collected from discharge cycles of Li-ion batteries cycled under various conditions are analyzed. The battery cycling data is sourced from a publically available repository; provided by Prognostics Center of Excellence (PCoE) at Ames Research Center, NASA [56]. Table 1 lists the 19 batteries used in this work, along with their respective operating

parameters. The data repository [56] contains capacity, voltage, current, temperature, current load and voltage load recorded for each discharge cycle.

- **Step 2:** Discharge cycle data segregation into classes.
- **Step 3:** Extract voltage , temperature , time parameters of each cycle.
- **Step 4:** Compute secondary parameters from voltage and temperature curves.
- **Step 5:** Prepare training dataset from primary and secondary data parameters.
- **Step 6:** Optimize feature vector for classification.
- **Step 7:** Apply machine learning framework SVM Method.
- **Step 8:** RUL Classifier Model

Thus, we have explained the complete methodology of our main ANN MODEL. We also explained the methodology of CNN Model which we are using for our comparative analysis. As well as the methodology of other models like linear regression and SVM.

Feature Extraction

Data collected from discharge cycles of Li-ion batteries cycled under various conditions are analyzed. The battery cycling data is provided by Prognostics Center of Excellence (PCoE) at Ames Research Center, NASA. The data repository contains capacity, voltage, current, temperature, current load and voltage load recorded for each discharge cycle of the batteries. Except the cell capacity, all other parameters are recorded over time during discharge; however these parameters are acquired with non-uniform sampling rate. It is observed that as the battery ages there will be change in measured voltage, current and temperature. Hence it is paramount to extract the relevant features from these curves that are crucial in determining battery life. From each discharge cycle, a set of 8 parameters is extracted from voltage and temperature curves representing minimum and maximum values of each curve, and their respective times. In addition to the above parameters, the following 13 parameters are computed from voltage, temperature and current curves for each discharge cycle.

- **Capacity (Cap):** The capacity of battery is computed by integrating discharge current over time and it is given by:

$$\text{Cap} = \int_{t_1}^{t_2} I dt$$

- **Energy of signal (E):** Signal energy of voltage and temperature curves are computed. In general, energy of signal is defined as the measure of signal strength over time and it is given by Eq. Below: where x(t) is the signal (either voltage or temperature) and t is time. In this work VCE notation is used to denote energy of voltage curve and TCE to denote energy of temperature curve.

$$E = \int_{-\infty}^{\infty} |x(t)|^2 dt$$

- **Fluctuation index of signal (FI):** Fluctuation Index of signal is defined as a measure of deviation of the signal from the mean and is given as:

$$\mathrm{FI} = \frac{\sqrt{\sum (y_i - \mu)^2}}{\omega}$$

where yi is the signal, μ is the mean of the signal and ω is sampling Frequency.

- **Skewness index (SI):** Skewness Index is a measure of the extent to which a probability distribution of signal leans toward the mean of the signal. This index is given by:

$$\mathrm{SI} = \frac{\sum_{i=1}^{n} (y_i - \mu)^3}{\sigma^3}$$

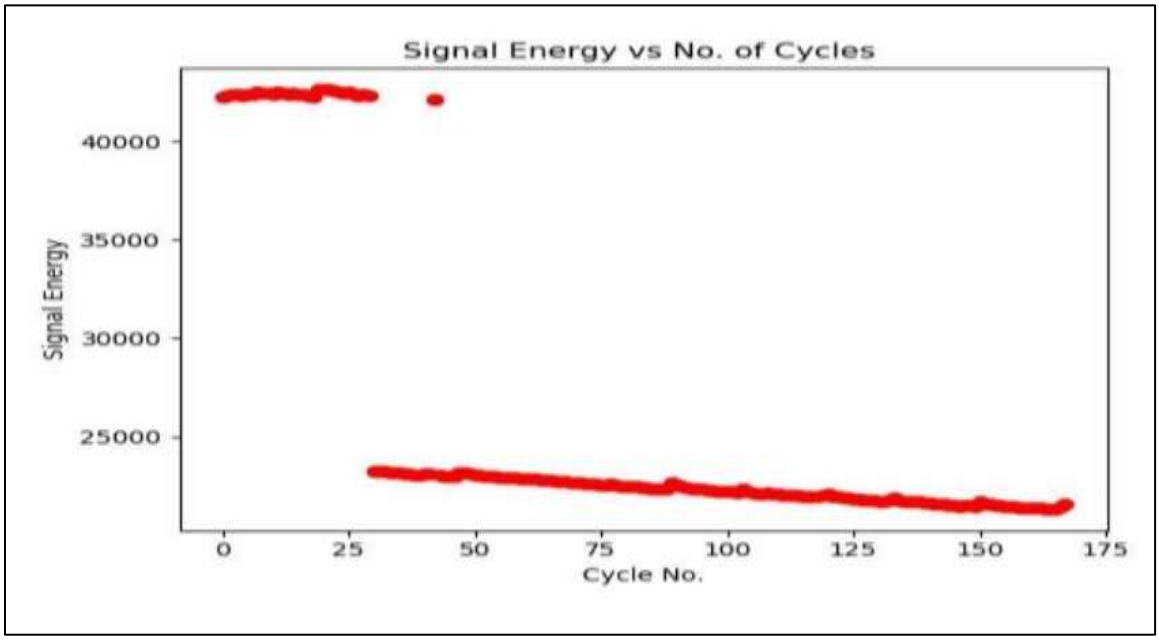
where yi is input signal l is mean of the signal and r is standard deviation of the signal and n is length of signal. Skewness index of the voltage curve is denoted by VC_SI and that of the temperature curve is denoted by TC_SI.

- **Kurtosis index (KI):** It is measure of the ‘‘peakedness” of the probability distribution of the signal and it is given by the equation:

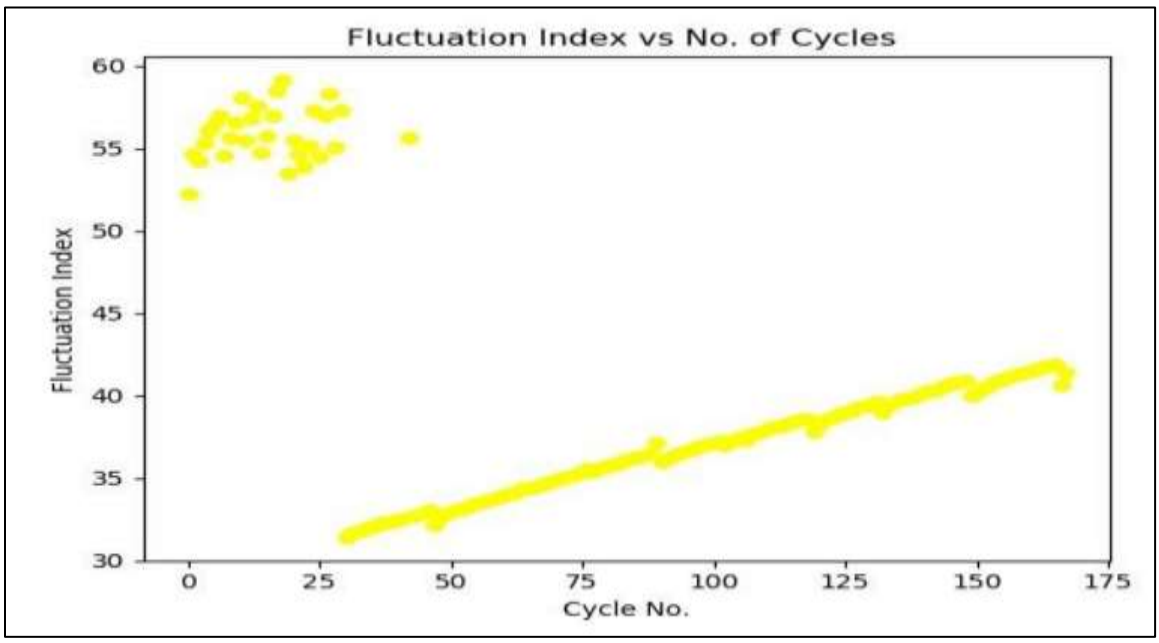
$$\mathrm{KI} = \frac{\sum_{i=1}^{n} (y_i - \mu)^4}{\sigma^4}$$

VC_KI and TC_KI notations used to denote the kurtosis index of voltage and temperature curves respectively.

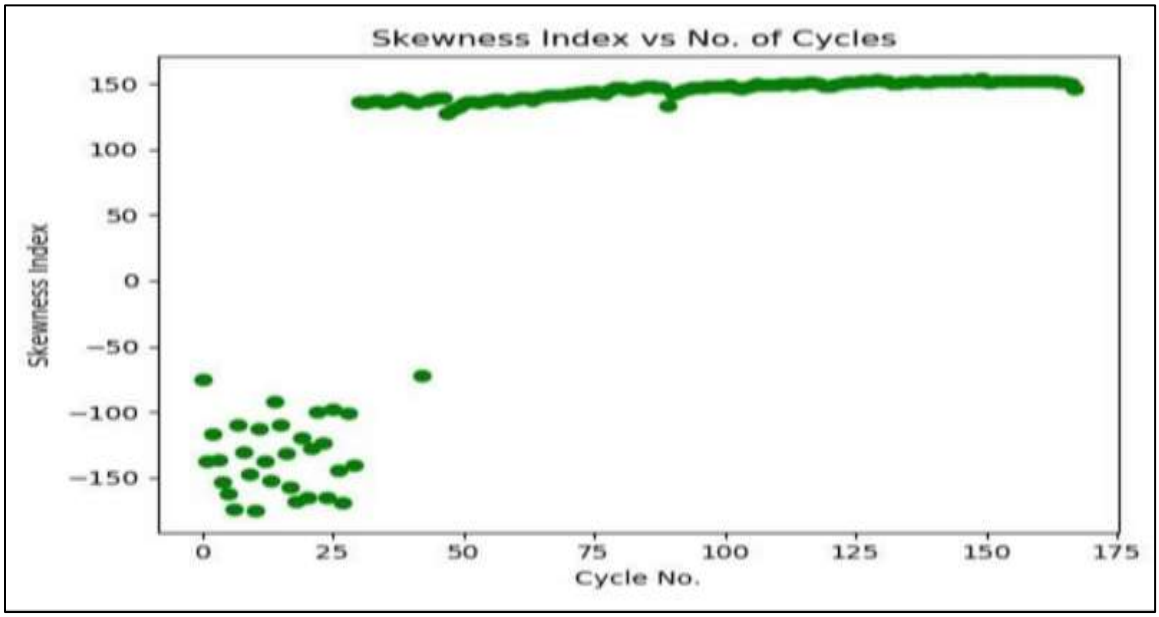
The graph below shows the signal energy and number of cycles . Initially the values of signal energy are constant But after the 25 th cycle, the signal energy drops and then goes on decreasing showing a specific pattern.



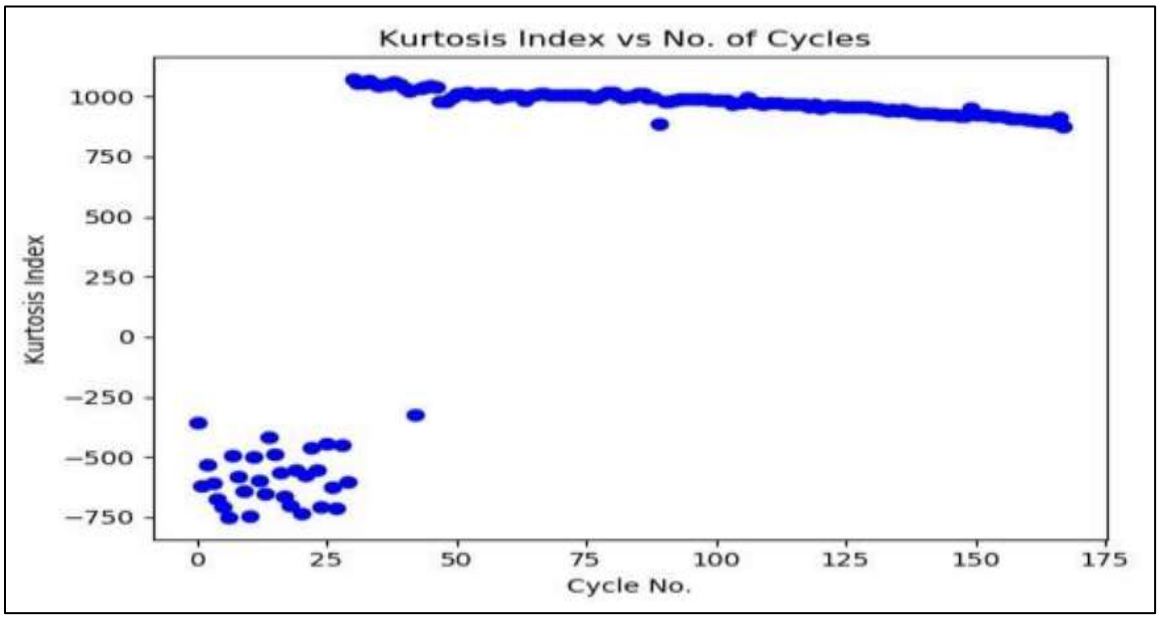
The graph below shows the fluctuation index and number of cycles . Initially the values of the fluctuation index are scattered. But after the 25th cycle the fluctuation index shows a sudden drop and then goes on increasing linearly with cycle number.



The graph below shows the skewness index and number of cycles . Initially the values of the skewness index are scattered. But after 25th cycle, the value increases suddenly and then remains constant



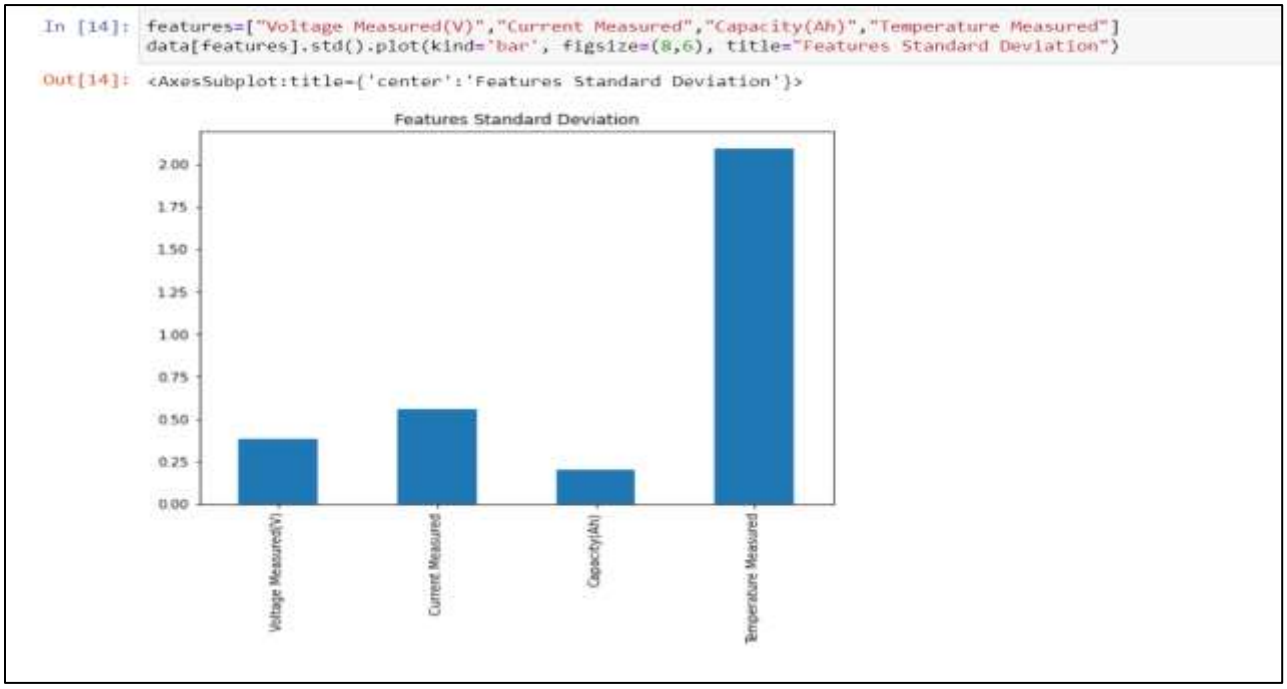
The graph below shows the kurtosis index and number of cycles for battery B0005 . Initially the values of the kurtosis index are negative and scattered. After the 50th cycle value of kurtosis index is positive and maintain almost a constant value.



The python code and the generated csv are attached with the document

Our Contribution

Our contribution is that we have created ANN Model from scratch for predicting the RUL of Lithium-Ion Batteries. The SGD is nothing but Stochastic Gradient Descent, It is an optimizer which comes under gradient descent which is an famous optimization technique used in machine learning and deep learning. The SGD optimizer in which "stochastic" means a system which is connected or linked up with random probability. In SGD optimizer a few samples is being picked up or we can say a few samples being get selected in a random manner instead taking up the whole dataset for each iteration. For optimizing a function we are going to use torch.optim which is a package, implements numerous optimization algorithms. The several commonly used methods are already supported, and the interface is general enough so that more practical ones can be also easily integrated in future. Our contribution involved creating a feature standard deviation graph for analyzing the dataset.



We wrote ANN model code from scratch while using different activation functions apart from the ones given in the research paper. We have used all three activation functions, sigma, tan-h ,and relu whereas the research paper has used only tan-h and linear activation function. We have used sgd optimizer, whereas in the research paper they used a bat filter, as sgd optimizer we could get 99% accuracy.

```
from keras.models import Sequential
from keras.layers import Dense
from tensorflow.keras.optimizers import Adam
from keras.layers import Dropout
from keras import regularizers

# define a function to build the keras model
def create_model():
    # create model
    model = Sequential()
    model.add(Dense(10, input_dim=5, kernel_initializer='uniform', activation='tanh'))
    model.add(Dropout(0.15))
    model.add(Dense(7, kernel_initializer='uniform', activation='relu'))
    model.add(Dropout(0.15))
    model.add(Dense(3,kernel_initializer='uniform', activation='sigmoid'))

    # compile model
    model.compile(loss='mean_squared_error', optimizer='sgd', metrics=['mean_absolute_error','accuracy'])
    return model

model = create_model()

print(model.summary())
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 10)                60

 dropout (Dropout)           (None, 10)                0

 dense_1 (Dense)             (None, 7)                 77

 dropout_1 (Dropout)         (None, 7)                 0

 dense_2 (Dense)             (None, 3)                 24

=================================================================
Total params: 161
Trainable params: 161
Non-trainable params: 0
_________________________________________________________________
None
```

Fitting the proposed model

In [24]:
```
history=model.fit(X_train, Y_train, validation_data=(X_test, Y_test
```

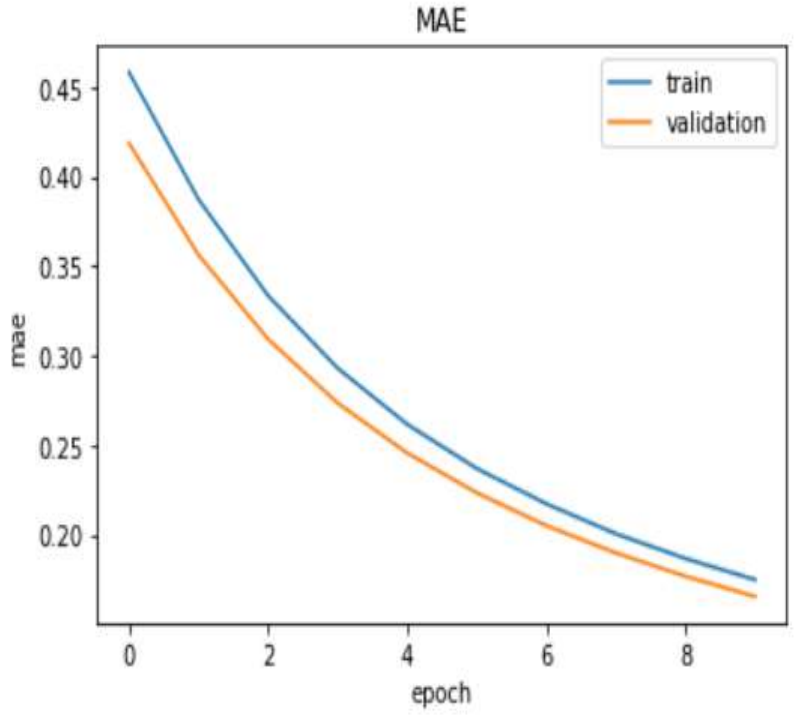
```
Epoch 1/10
445/445 [==============================] - 11s 6ms/step - loss:
0.2107 - mean_absolute_error: 0.4583 - accuracy: 0.9865 - val_los
s: 0.1754 - val_mean_absolute_error: 0.4187 - val_accuracy: 0.994
8
Epoch 2/10
445/445 [==============================] - 2s 5ms/step - loss: 0.
1508 - mean_absolute_error: 0.3873 - accuracy: 0.9865 - val_loss:
0.1272 - val_mean_absolute_error: 0.3562 - val_accuracy: 0.9948
Epoch 3/10
445/445 [==============================] - 2s 4ms/step - loss: 0.
1125 - mean_absolute_error: 0.3335 - accuracy: 0.9865 - val_loss:
0.0962 - val_mean_absolute_error: 0.3093 - val_accuracy: 0.9948
Epoch 4/10
445/445 [==============================] - 2s 4ms/step - loss: 0.
0875 - mean_absolute_error: 0.2930 - accuracy: 0.9865 - val_loss:
0.0755 - val_mean_absolute_error: 0.2735 - val_accuracy: 0.9948
Epoch 5/10
445/445 [==============================] - 2s 3ms/step - loss: 0.
0706 - mean_absolute_error: 0.2617 - accuracy: 0.9865 - val_loss:
0.0613 - val_mean_absolute_error: 0.2457 - val_accuracy: 0.9948
Epoch 6/10
445/445 [==============================] - 2s 3ms/step - loss: 0.
0588 - mean_absolute_error: 0.2370 - accuracy: 0.9865 - val_loss:
0.0510 - val_mean_absolute_error: 0.2234 - val_accuracy: 0.9948
```

```
Epoch 7/10
445/445 [==============================] - 2s 3ms/step - loss: 0.
0501 - mean_absolute_error: 0.2173 - accuracy: 0.9865 - val_loss:
0.0433 - val_mean_absolute_error: 0.2051 - val_accuracy: 0.9948
Epoch 8/10
445/445 [==============================] - 2s 3ms/step - loss: 0.
0435 - mean_absolute_error: 0.2006 - accuracy: 0.9865 - val_loss:
0.0374 - val_mean_absolute_error: 0.1898 - val_accuracy: 0.9948
Epoch 9/10
445/445 [==============================] - 2s 4ms/step - loss: 0.
0385 - mean_absolute_error: 0.1867 - accuracy: 0.9865 - val_loss:
0.0327 - val_mean_absolute_error: 0.1768 - val_accuracy: 0.9948
Epoch 10/10
445/445 [==============================] - 2s 4ms/step - loss: 0.
0345 - mean_absolute_error: 0.1749 - accuracy: 0.9865 - val_loss:
0.0290 - val_mean_absolute_error: 0.1656 - val_accuracy: 0.9948
```

We have Plotted the obtained and predicted Mean Absolute Error (MAE) by combining all attributes. Mean absolute error (MAE) is a measure of errors between paired observations expressing the same phenomenon. Examples of Y versus X include comparisons of predicted versus observed, subsequent time versus initial time, and one technique of measurement versus an alternative technique of measurement.

Plotting the obtained and predicted Mean Absolute Error (MAE)

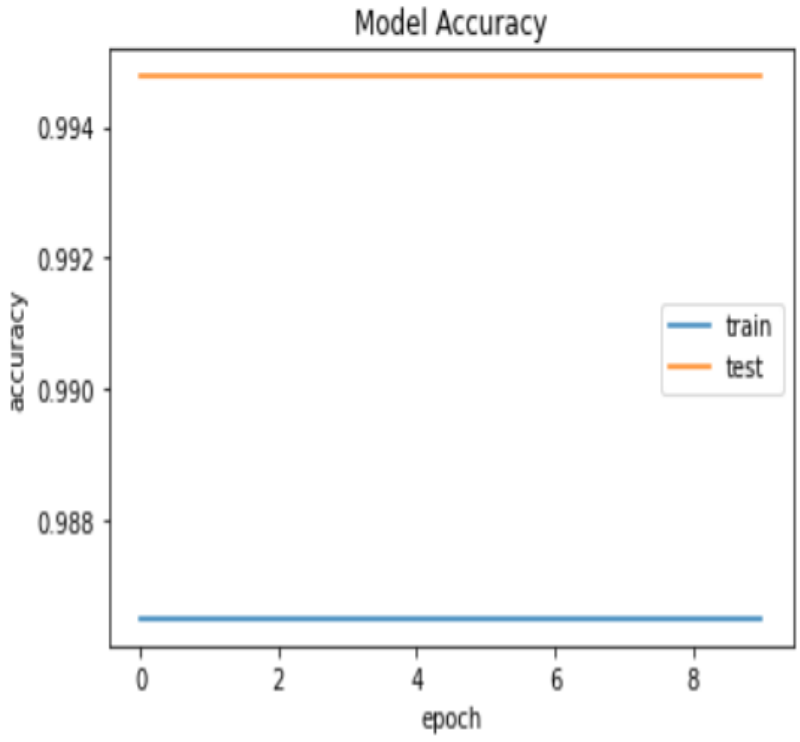
In [26]:
```
plt.plot(history.history['mean_absolute_error'])
plt.plot(history.history['val_mean_absolute_error'])
plt.title('MAE')
plt.ylabel('mae')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper right')
plt.show()
```



We have a Plotted graph of the obtained and predicted Accuracy by combining all attributes.The epoch and accuracy data of the model for each epoch is stored in the history object. The below snippet plots the graph of the training loss vs. validation loss over the number of epochs.

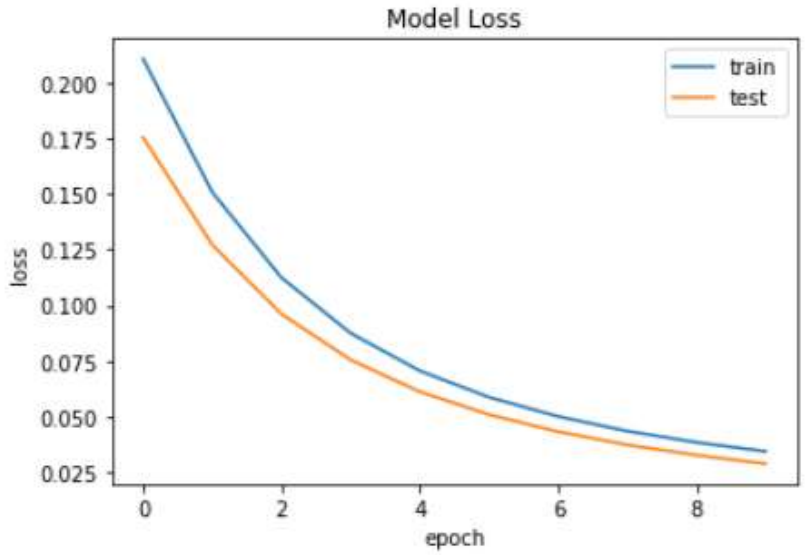
Plotting the obtained and predicted Accuracy

In [27]:
```
import matplotlib.pyplot as plt
%matplotlib inline
# Model accuracy
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model Accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'])
plt.show()
```



We are Plotting the obtained and predicted Mean Squared Error (MSE) by combining all attributes.In statistics, the concept of mean squared error is an essential measure utilized to determine the performance of an estimator. It is abbreviated as MSE and is necessary for relaying the concepts of precision, bias and accuracy during the statistical estimation.

```
In [28]: plt.plot(history.history['loss'])
         plt.plot(history.history['val_loss'])
         plt.title('Model Loss')
         plt.ylabel('loss')
         plt.xlabel('epoch')
         plt.legend(['train', 'test'])
         plt.show()
```



Plotting the graph of Cycle Vs Capacity

Graphs and Inferences: RUL Prediction using SGD optimizer:

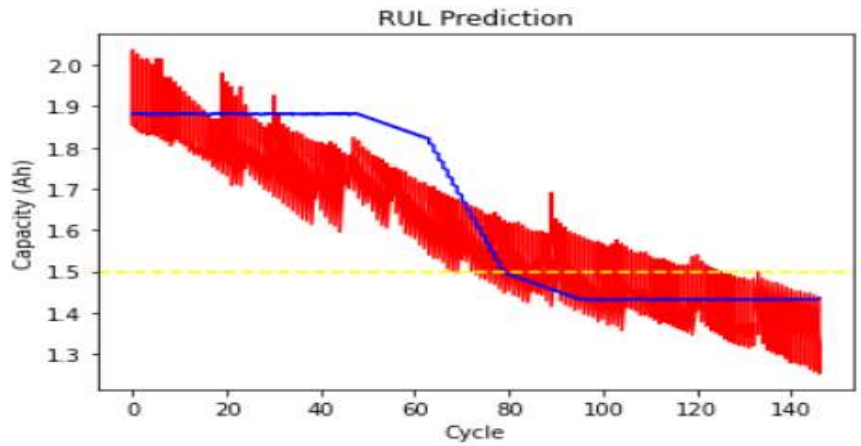
Details of Neural Network:

- Input Layer: 1 neuron, Activation function= tanh
- Two Hidden layers: Sigmoid and RELU are hidden layers, along with 2 other dense functions.
- Input to the neural network : Cycle No.

- Output of the network : Capacity
- Optimizer : SGD Loss: Mean squared error

The Red line indicates capacity values for the battery dataset. The Blue line indicates the predicted values using NN (optimizer=sgd). The Neural network was trained on Batteries B0006, B0007, B0018 from the dataset provided by NASA However it is observed from the trails performed during implementation that prediction based on only cycle number does give accurate results.

```
y_prediction = [*f, *e, *d]
import matplotlib.pyplot as plt
threshold=1.5
#X_Cycle = X_test[:,0]
X_Cycle = np.array(data.iloc[:573,0].values)
Y_Capacity = np.array(data.iloc[:573,5].values)
plt.plot(X_Cycle, Y_Capacity, color='red')
plt.plot(X_Cycle, y_prediction, color='blue')
plt.axhline(threshold, color='yellow', linestyle='--')
plt.title('RUL Prediction')
plt.xlabel('Cycle')
plt.ylabel('Capacity (Ah)')
plt.show()
```

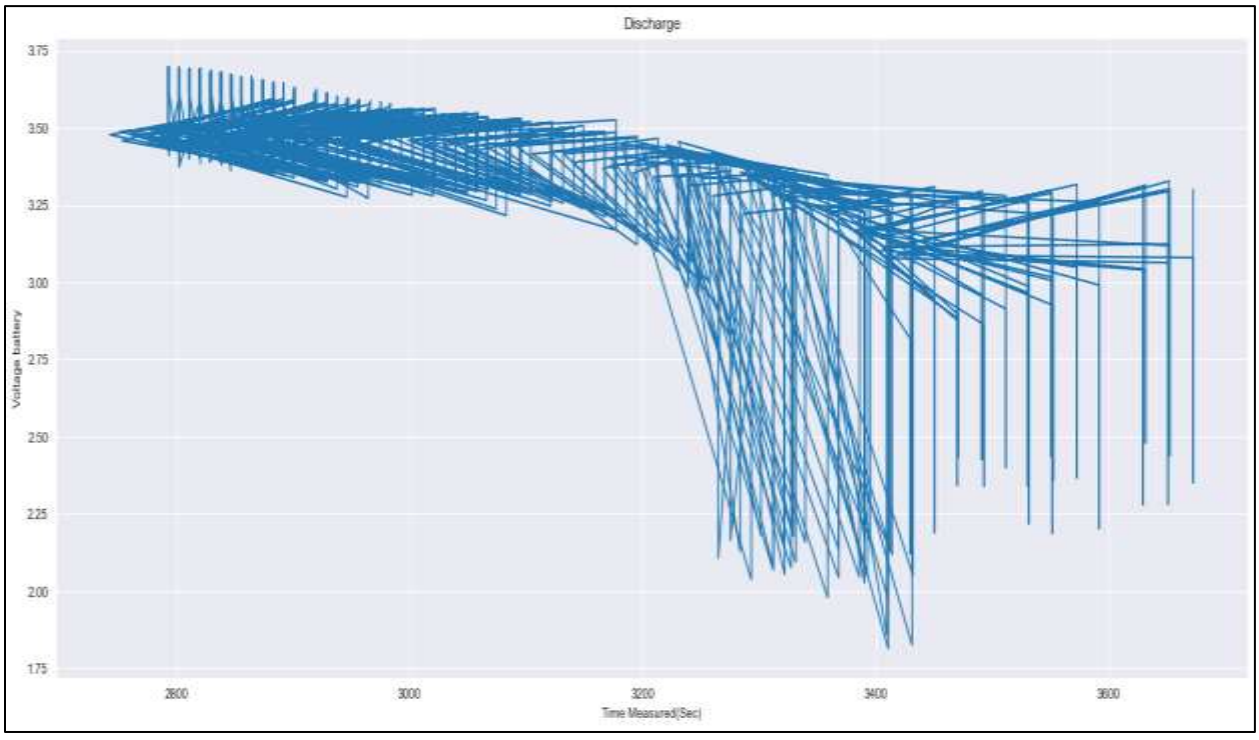


Plotting the graph of Time Measured Vs Voltage Measured for all cycles which gives us the discharge parameter of graphs which is essential in rul prediction while analyzing the dataset of rul prediction.

Plotting the graph of Time Measured Vs Voltage Measured for all cycles

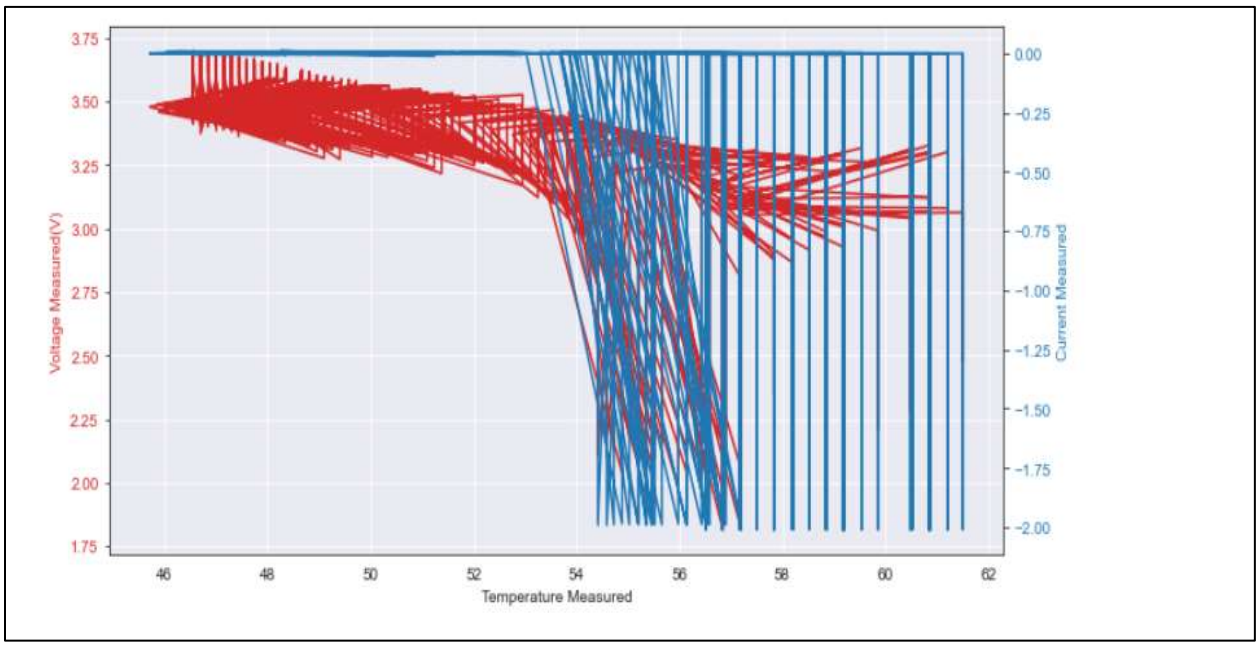
In [30]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns

plot_df = data.loc[(data['Cycle']>=1),['Time Measured(Sec)','Voltage Measured(V)']]
sns.set_style("darkgrid")
plt.figure(figsize=(20, 8))
plt.plot(plot_df['Time Measured(Sec)'], plot_df['Voltage Measured(V)'])
plt.ylabel('Voltage battery')

# make x-axis ticks legible
adf = plt.gca().get_xaxis().get_major_formatter()
#adf.scaled[1.0] = '%m-%d-%Y'
plt.xlabel('Time Measured(Sec)')
plt.title('Discharge')
```

Out[30]: Text(0.5, 1.0, 'Discharge')



Plotting the observed relationship between Temperature Measured, Voltage Measured and Current Measured. The voltage , temperature measured and current measured are essential parameters in prediction for rul of lithium ion batteries and these parameters are analyzed through this graph for us to draw further conclusions and inferences from it.

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
# Create some mock data

fig, ax1 = plt.subplots()
sns.set_style("white")
plot_df= data.loc[(data['Cycle']==1),['Temperature Measured','Current Measured']]
#plt.plot([126, 127], color="black")
plot_df1 = data.loc[(data['Cycle']==1),['Temperature Measured','Voltage Measured(V)']]

color = 'tab:red'
ax1.set_xlabel('Temperature Measured')
ax1.set_ylabel('Voltage Measured(V)', color=color)
ax1.plot(data['Time Measured(Sec)']/60, data['Voltage Measured(V)'], color=color)
ax1.tick_params(axis='y', labelcolor=color)

ax2 = ax1.twinx()  # instantiate a second axes that shares the same x-axis

color = 'tab:blue'
ax2.set_ylabel('Current Measured', color=color)  # we already handled the x-label with ax1
ax2.plot(data['Time Measured(Sec)']/60, data['Current Measured'],'-', color=color)
ax2.tick_params(axis='y', labelcolor=color)
fig.set_size_inches(10, 5)
fig.tight_layout()  # otherwise the right y-label is slightly clipped
plt.show()
# vertical black line split the graph between charge and discharge operation.
```



NEW TESTS:

An attempt was made to predict RUL using NN for Input cycle number, Time measured, Voltage measured, current measured and temperature measured. To extrapolate the NN, all other features except the cycle number were to be determined. To predict the features neural networks were applied. After applying neural networks on Cycle number versus each of the above mentioned features, following values of accuracy and loss graph is obtained.

Basically , we are trying to use all attributes of our dataset to see if accuracy is further improving and how it's impacting our ANN model and whether we can use it for RUL Prediction.

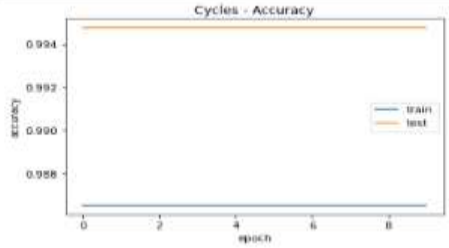
- **First Attribute: Cycle**

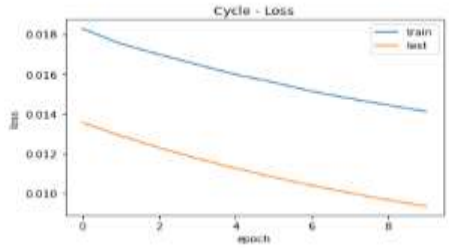
```
Cycle_X = np.array(data.iloc[:,0].values)
y = np.array(data.iloc[:,5].values)
Cycle_X = Cycle_X.astype('float32')
mean = Cycle_X.mean(axis=0)
Cycle_X -= mean
std = Cycle_X.std(axis=0)
Cycle_X /= std

# create X and Y datasets for training
from sklearn import model_selection

X_train, X_test, y_train, y_test = model_selection.train_test_split(Cycle_X, y, random_state=42, test_size = 0.3)

from keras.utils.np_utils import to_categorical

Y_train = to_categorical(y_train, num_classes=None)
Y_test = to_categorical(y_test, num_classes=None)
print (Y_train.shape)
print (Y_train[:10])

history=model.fit(X_train, Y_train, validation_data=(X_test, Y_test),epochs=10, batch_size=1)
```

```
(445, 3)
[[0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]]
Epoch 1/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0325 - mean_absolute_error: 0.1687 - accuracy: 0.9865 - val_lo
ss: 0.0274 - val_mean_absolute_error: 0.1606 - val_accuracy: 0.9948
Epoch 2/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0299 - mean_absolute_error: 0.1602 - accuracy: 0.9865 - val_lo
ss: 0.0249 - val_mean_absolute_error: 0.1524 - val_accuracy: 0.9948
Epoch 3/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0277 - mean_absolute_error: 0.1526 - accuracy: 0.9865 - val_lo
ss: 0.0229 - val_mean_absolute_error: 0.1452 - val_accuracy: 0.9948
Epoch 4/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0259 - mean_absolute_error: 0.1460 - accuracy: 0.9865 - val_lo
ss: 0.0211 - val_mean_absolute_error: 0.1387 - val_accuracy: 0.9948
Epoch 5/10
445/445 [==============================] - 2s 3ms/step - loss: 0.0244 - mean_absolute_error: 0.1400 - accuracy: 0.9865 - val_lo
ss: 0.0195 - val_mean_absolute_error: 0.1328 - val_accuracy: 0.9948
Epoch 6/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0229 - mean_absolute_error: 0.1342 - accuracy: 0.9865 - val_lo
ss: 0.0182 - val_mean_absolute_error: 0.1275 - val_accuracy: 0.9948
Epoch 7/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0218 - mean_absolute_error: 0.1293 - accuracy: 0.9865 - val_lo
ss: 0.0170 - val_mean_absolute_error: 0.1227 - val_accuracy: 0.9948
Epoch 8/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0207 - mean_absolute_error: 0.1244 - accuracy: 0.9865 - val_lo
ss: 0.0160 - val_mean_absolute_error: 0.1182 - val_accuracy: 0.9948
Epoch 9/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0198 - mean_absolute_error: 0.1203 - accuracy: 0.9865 - val_lo
ss: 0.0151 - val_mean_absolute_error: 0.1140 - val_accuracy: 0.9948
Epoch 10/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0190 - mean_absolute_error: 0.1165 - accuracy: 0.9865 - val_lo
ss: 0.0143 - val_mean_absolute_error: 0.1102 - val_accuracy: 0.9948
```

In [31]:

```
import matplotlib.pyplot as plt
%matplotlib inline
# Model accuracy
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Cycles - Accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'])
plt.show()

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Cycle - Loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'])
plt.show()
```



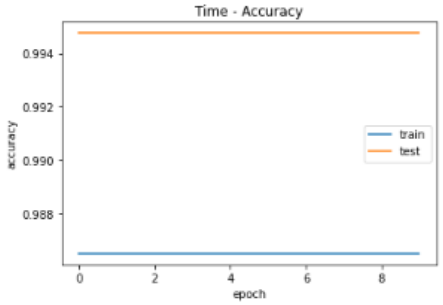


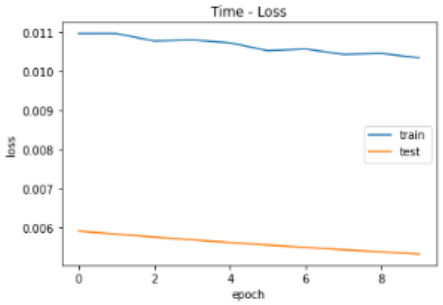
- **Second Attribute: Time Measured(Sec)**

```
Time_X = np.array(data.iloc[:,1].values)
Time_X = Time_X.astype('float32')
y = np.array(data.iloc[:,5].values)
mean = Time_X.mean(axis=0)
Time_X -= mean
std = Time_X.std(axis=0)
Time_X /= std

# create X and Y datasets for training
from sklearn import model_selection

X_train, X_test, y_train, y_test = model_selection.train_test_split(Time_X, y, random_state=42, test_size = 0.3)

from keras.utils.np_utils import to_categorical

Y_train = to_categorical(y_train, num_classes=None)
Y_test = to_categorical(y_test, num_classes=None)
print (Y_train.shape)
print (Y_train[:10])

history=model.fit(X_train, Y_train, validation_data=(X_test, Y_test),epochs=10, batch_size=1)
```

```
(445, 3)
[[0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]]
Epoch 1/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0110 - mean_absolute_error: 0.0619 - accuracy: 0.9865 - val_lo
ss: 0.0059 - val_mean_absolute_error: 0.0558 - val_accuracy: 0.9948
Epoch 2/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0110 - mean_absolute_error: 0.0611 - accuracy: 0.9865 - val_lo
ss: 0.0058 - val_mean_absolute_error: 0.0550 - val_accuracy: 0.9948
Epoch 3/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0108 - mean_absolute_error: 0.0596 - accuracy: 0.9865 - val_lo
ss: 0.0057 - val_mean_absolute_error: 0.0542 - val_accuracy: 0.9948
Epoch 4/10
445/445 [==============================] - 2s 3ms/step - loss: 0.0108 - mean_absolute_error: 0.0598 - accuracy: 0.9865 - val_lo
ss: 0.0057 - val_mean_absolute_error: 0.0535 - val_accuracy: 0.9948
Epoch 5/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0107 - mean_absolute_error: 0.0590 - accuracy: 0.9865 - val_lo
ss: 0.0056 - val_mean_absolute_error: 0.0527 - val_accuracy: 0.9948
Epoch 6/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0105 - mean_absolute_error: 0.0575 - accuracy: 0.9865 - val_lo
ss: 0.0055 - val_mean_absolute_error: 0.0520 - val_accuracy: 0.9948
Epoch 7/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0106 - mean_absolute_error: 0.0566 - accuracy: 0.9865 - val_lo
ss: 0.0055 - val_mean_absolute_error: 0.0514 - val_accuracy: 0.9948
Epoch 8/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0104 - mean_absolute_error: 0.0562 - accuracy: 0.9865 - val_lo
ss: 0.0054 - val_mean_absolute_error: 0.0507 - val_accuracy: 0.9948
Epoch 9/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0104 - mean_absolute_error: 0.0556 - accuracy: 0.9865 - val_lo
ss: 0.0054 - val_mean_absolute_error: 0.0501 - val_accuracy: 0.9948
Epoch 10/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0103 - mean_absolute_error: 0.0548 - accuracy: 0.9865 - val_lo
ss: 0.0053 - val_mean_absolute_error: 0.0495 - val_accuracy: 0.9948
```

In [37]:
```
import matplotlib.pyplot as plt
%matplotlib inline
# Model accuracy
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Time - Accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'])
plt.show()

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Time - Loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'])
plt.show()
```



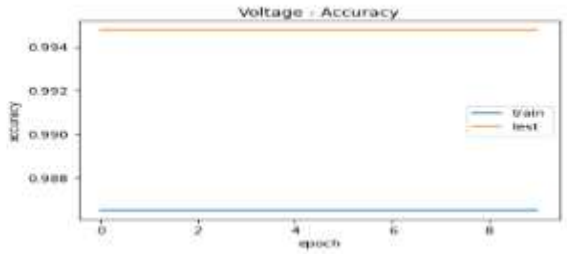


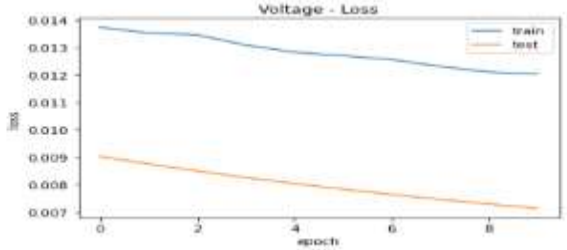
- **Third Attribute: Voltage Measured(V)**

```
Voltage_X = np.array(data.iloc[:,2].values)
Voltage_X = Voltage_X.astype('float32')
y = np.array(data.iloc[:,5].values)
mean = Voltage_X.mean(axis=0)
Voltage_X -= mean
std = Voltage_X.std(axis=0)
Voltage_X /= std

# create X and Y datasets for training
from sklearn import model_selection

X_train, X_test, y_train, y_test = model_selection.train_test_split(Voltage_X, y, random_state=42, test_size = 0.3)

from keras.utils.np_utils import to_categorical

Y_train = to_categorical(y_train, num_classes=None)
Y_test = to_categorical(y_test, num_classes=None)
print (Y_train.shape)
print (Y_train[:10])

history=model.fit(X_train, Y_train, validation_data=(X_test, Y_test),epochs=10, batch_size=1)
```

```
(445, 3)
[[0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]]
Epoch 1/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0137 - mean_absolute_error: 0.0863 - accuracy: 0.9865 - val_lo
ss: 0.0090 - val_mean_absolute_error: 0.0809 - val_accuracy: 0.9948
Epoch 2/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0135 - mean_absolute_error: 0.0848 - accuracy: 0.9865 - val_lo
ss: 0.0087 - val_mean_absolute_error: 0.0791 - val_accuracy: 0.9948
Epoch 3/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0135 - mean_absolute_error: 0.0835 - accuracy: 0.9865 - val_lo
ss: 0.0085 - val_mean_absolute_error: 0.0773 - val_accuracy: 0.9948
Epoch 4/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0131 - mean_absolute_error: 0.0812 - accuracy: 0.9865 - val_lo
ss: 0.0083 - val_mean_absolute_error: 0.0756 - val_accuracy: 0.9948
Epoch 5/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0128 - mean_absolute_error: 0.0793 - accuracy: 0.9865 - val_lo
ss: 0.0080 - val_mean_absolute_error: 0.0740 - val_accuracy: 0.9948
Epoch 6/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0127 - mean_absolute_error: 0.0779 - accuracy: 0.9865 - val_lo
ss: 0.0078 - val_mean_absolute_error: 0.0725 - val_accuracy: 0.9948
Epoch 7/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0126 - mean_absolute_error: 0.0769 - accuracy: 0.9865 - val_lo
ss: 0.0076 - val_mean_absolute_error: 0.0710 - val_accuracy: 0.9948
Epoch 8/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0123 - mean_absolute_error: 0.0749 - accuracy: 0.9865 - val_lo
ss: 0.0075 - val_mean_absolute_error: 0.0696 - val_accuracy: 0.9948
Epoch 9/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0121 - mean_absolute_error: 0.0736 - accuracy: 0.9865 - val_lo
ss: 0.0073 - val_mean_absolute_error: 0.0683 - val_accuracy: 0.9948
Epoch 10/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0120 - mean_absolute_error: 0.0724 - accuracy: 0.9865 - val_lo
ss: 0.0071 - val_mean_absolute_error: 0.0670 - val_accuracy: 0.9948
```

```
import matplotlib.pyplot as plt
%matplotlib inline
# Model accuracy
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Voltage - Accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'])
plt.show()

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Voltage - Loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'])
plt.show()
```



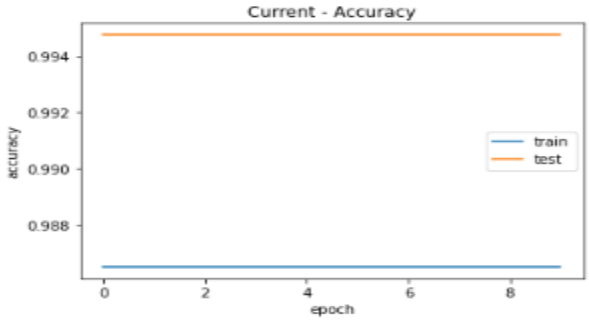


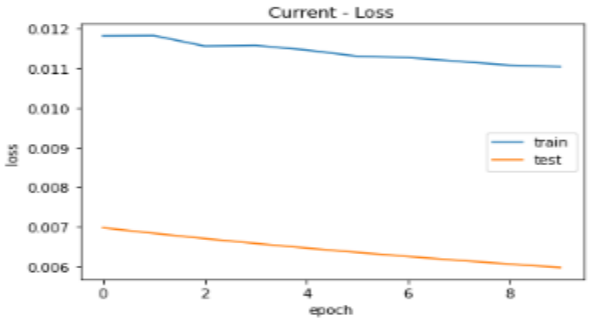
- **Fourth Attribute: Current Measured**

```
Current_X = np.array(data.iloc[:,3].values)
Current_X = Current_X.astype('float32')
y = np.array(data.iloc[:,5].values)
mean = Current_X.mean(axis=0)
Current_X -= mean
std = Current_X.std(axis=0)
Current_X /= std

# create X and Y datasets for training
from sklearn import model_selection

X_train, X_test, y_train, y_test = model_selection.train_test_split(Current_X, y, random_state=42, test_size = 0.3)

from keras.utils.np_utils import to_categorical

Y_train = to_categorical(y_train, num_classes=None)
Y_test = to_categorical(y_test, num_classes=None)
print (Y_train.shape)
print (Y_train[:10])

history=model.fit(X_train, Y_train, validation_data=(X_test, Y_test),epochs=10, batch_size=1)
```

```
(445, 3)
[[0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]]
Epoch 1/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0118 - mean_absolute_error: 0.0713 - accuracy: 0.9865 - val_lo
ss: 0.0070 - val_mean_absolute_error: 0.0657 - val_accuracy: 0.9948
Epoch 2/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0118 - mean_absolute_error: 0.0707 - accuracy: 0.9865 - val_lo
ss: 0.0068 - val_mean_absolute_error: 0.0646 - val_accuracy: 0.9948
Epoch 3/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0116 - mean_absolute_error: 0.0687 - accuracy: 0.9865 - val_lo
ss: 0.0067 - val_mean_absolute_error: 0.0634 - val_accuracy: 0.9948
Epoch 4/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0116 - mean_absolute_error: 0.0681 - accuracy: 0.9865 - val_lo
ss: 0.0066 - val_mean_absolute_error: 0.0623 - val_accuracy: 0.9948
Epoch 5/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0115 - mean_absolute_error: 0.0673 - accuracy: 0.9865 - val_lo
ss: 0.0065 - val_mean_absolute_error: 0.0613 - val_accuracy: 0.9948
Epoch 6/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0113 - mean_absolute_error: 0.0656 - accuracy: 0.9865 - val_lo
ss: 0.0064 - val_mean_absolute_error: 0.0603 - val_accuracy: 0.9948
Epoch 7/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0113 - mean_absolute_error: 0.0649 - accuracy: 0.9865 - val_lo
ss: 0.0063 - val_mean_absolute_error: 0.0593 - val_accuracy: 0.9948
Epoch 8/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0112 - mean_absolute_error: 0.0643 - accuracy: 0.9865 - val_lo
ss: 0.0062 - val_mean_absolute_error: 0.0584 - val_accuracy: 0.9948
Epoch 9/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0111 - mean_absolute_error: 0.0632 - accuracy: 0.9865 - val_lo
ss: 0.0061 - val_mean_absolute_error: 0.0575 - val_accuracy: 0.9948
Epoch 10/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0110 - mean_absolute_error: 0.0626 - accuracy: 0.9865 - val_lo
ss: 0.0060 - val_mean_absolute_error: 0.0566 - val_accuracy: 0.9948
```

```
import matplotlib.pyplot as plt
%matplotlib inline
# Model accuracy
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Current - Accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'])
plt.show()

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Current - Loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'])
plt.show()
```



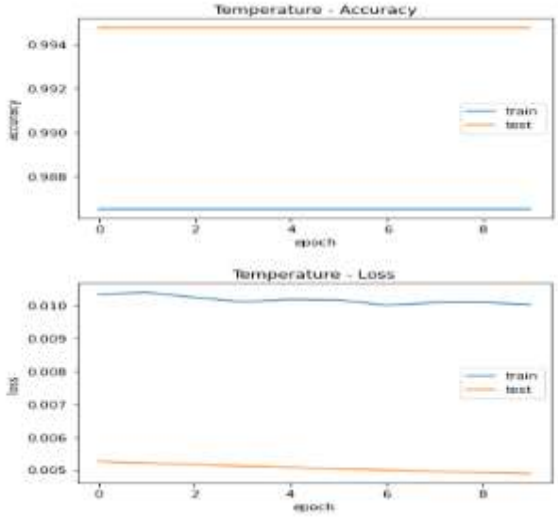


- **Fifth Attribute: Temperature Measured**

```
Temperature_X = np.array(data.iloc[:,3].values)
Temperature_X = Temperature_X.astype('float32')
y = np.array(data.iloc[:,5].values)
mean = Temperature_X.mean(axis=0)
Temperature_X -= mean
std = Temperature_X.std(axis=0)
Temperature_X /= std

# create X and Y datasets for training
from sklearn import model_selection

X_train, X_test, y_train, y_test = model_selection.train_test_split(Temperature_X, y, random_state=42, test_size = 0.3)

from keras.utils.np_utils import to_categorical

Y_train = to_categorical(y_train, num_classes=None)
Y_test = to_categorical(y_test, num_classes=None)
print (Y_train.shape)
print (Y_train[:10])

history=model.fit(X_train, Y_train, validation_data=(X_test, Y_test),epochs=10, batch_size=1)
```

```
(445, 3)
[[0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]]
Epoch 1/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0103 - mean_absolute_error: 0.0543 - accuracy: 0.9865 - val_lo
ss: 0.0053 - val_mean_absolute_error: 0.0489 - val_accuracy: 0.9948
Epoch 2/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0104 - mean_absolute_error: 0.0544 - accuracy: 0.9865 - val_lo
ss: 0.0052 - val_mean_absolute_error: 0.0483 - val_accuracy: 0.9948
Epoch 3/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0102 - mean_absolute_error: 0.0531 - accuracy: 0.9865 - val_lo
ss: 0.0052 - val_mean_absolute_error: 0.0477 - val_accuracy: 0.9948
Epoch 4/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0101 - mean_absolute_error: 0.0525 - accuracy: 0.9865 - val_lo
ss: 0.0051 - val_mean_absolute_error: 0.0472 - val_accuracy: 0.9948
Epoch 5/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0102 - mean_absolute_error: 0.0525 - accuracy: 0.9865 - val_lo
ss: 0.0051 - val_mean_absolute_error: 0.0467 - val_accuracy: 0.9948
Epoch 6/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0102 - mean_absolute_error: 0.0523 - accuracy: 0.9865 - val_lo
ss: 0.0050 - val_mean_absolute_error: 0.0461 - val_accuracy: 0.9948
Epoch 7/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0100 - mean_absolute_error: 0.0511 - accuracy: 0.9865 - val_lo
ss: 0.0050 - val_mean_absolute_error: 0.0456 - val_accuracy: 0.9948
Epoch 8/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0101 - mean_absolute_error: 0.0506 - accuracy: 0.9865 - val_lo
ss: 0.0049 - val_mean_absolute_error: 0.0451 - val_accuracy: 0.9948
Epoch 9/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0101 - mean_absolute_error: 0.0507 - accuracy: 0.9865 - val_lo
ss: 0.0049 - val_mean_absolute_error: 0.0447 - val_accuracy: 0.9948
Epoch 10/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0100 - mean_absolute_error: 0.0501 - accuracy: 0.9865 - val_lo
ss: 0.0049 - val_mean_absolute_error: 0.0442 - val_accuracy: 0.9948
```

```
import matplotlib.pyplot as plt
%matplotlib inline
# Model accuracy
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Temperature - Accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'])
plt.show()

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Temperature - Loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'])
plt.show()
```



CONCLUSION AND INFERENCES DRAWN FROM THE ABOVE GRAPHS :

As we can see, there is a similar trend in accuracy over all attributes. It doesn't show great variations. However, the matter of concern is the error:

1. **Cycle:** This attribute's slopes pertaining to train & test's errors are similar. Their difference is also observed to be the least among all other attributes.
2. **Time:** The errors generated in the training period show an irregular line. The difference too is greater than the previous attribute, roughly 0.005. But the range of error is much smaller now.
3. **Voltage:** This attribute shows similar representations to the Time attribute, but the range of error is higher. The difference of 0.005 is now consistent over the remaining attributes as per our observation
4. **Current:** This attribute shows similar characteristics too. The difference is consistent but the range is lowered by 0.002.
5. **Temperature:** This attribute too includes similar features but is of utmost importance, as we can observe that even though the difference of 0.005 is present, we obtain the lowest range of mean squared error.

As observed, two attributes stand out, Cycles which show the least difference in mean square error in training and testing and hence provide accurate results & Temperature which drops the loss range to the lowest range. Concluding with the above observations, Cycles and Temperature play vital roles in the degradation and calculation of RUL in regard to Li-ion batteries. Our model works with 99% accuracy for independent variables too as shown in the explanation above.

Pseudocode

1. **READ** the dataset.
2. **SPLIT** the dataset into X and Y attributes for measurement values and target values
3. **STANDARDIZE** X attributes to the level its values.
4. **CREATE** train and test datasets.
5. **CREATE** a function to train the model:

```
DEFINE FUNCTION create_model():
        SET model = Sequential()
        model.add(Dense(10, input_dim=5,  kernel_initializer='uniform',
activation='tanh'))
        model.add(Dropout(0.15))
        model.add(Dense(7, kernel_initializer='uniform', activation='relu'))
        model.add(Dropout(0.15))
        model.add(Dense(3,kernel_initializer='uniform', activation='sigmoid'))
        model.compile(loss='mean_squared_error', optimizer='sgd',
metrics=['mean_absolute_error','accuracy'])
RETURN model
SET model = create_model()
```

6. **FIT** the proposed model.

SET history= model.fit(X_train, Y_train, validation_data=(X_test, Y_test),epochs=10, batch_size=1)

7. **PRINT** Train Loss, Train Mean Absolute Error, Train Accuracy.
8. **PRINT** Test Loss, Test Mean Absolute Error, Test Accuracy.
9. **PLOT** the obtained and predicted Mean Absolute Error (MAE) graph for train and test datasets.
10. **PLOT** the obtained and predicted Accuracy graph for train and test datasets.
11. **PLOT** the obtained and predicted Mean Squared Error (MSE) graph for train and test datasets.
12. **PLOT** the graph of Cycle Vs Capacity.
13. **PLOT** the Time Measured Vs Voltage Measured graph for all cycles.
14. **PLOT** the observed relationship between Temperature Measured, Voltage Measured and Current Measured.

Output

ANN MODEL:

The proposed ANN method is validated using the data collected by the Prognostics Center of Excellence at Ames Research Center, NASA. Experiment results show that the proposed ANN method can produce satisfactory RUL prediction results, which will assist condition-based maintenance optimization.

Importing the necessary libraries

The library NumPy is mainly used for linear algebra. The library pandas is used mainly for data processing, reading CSV file I/O (e.g. pd.read_csv).

```
In [1]: import sys
        import pandas as pd
        import numpy as np
        import sklearn
        import matplotlib
        import keras

        print('Python: {}'.format(sys.version))
        print('Pandas: {}'.format(pd.__version__))
        print('Numpy: {}'.format(np.__version__))
        print('Sklearn: {}'.format(sklearn.__version__))
        print('Matplotlib: {}'.format(matplotlib.__version__))
        print('Keras: {}'.format(keras.__version__))
```

```
Python: 3.10.2 (tags/v3.10.2:a58ebcc, Jan 17 2022, 14:12:15) [MSC v.1929 64 bit (AMD64)]
Pandas: 1.4.0
Numpy: 1.22.3
Sklearn: 1.0.2
Matplotlib: 3.5.1
Keras: 2.8.0
```

MatPlotLib Library will be useful for visualisations and to make observations

MatPlotLib Library will be useful for visualizations and to make observations.

```
In [2]: import matplotlib.pyplot as plt
        from pandas.plotting import scatter_matrix
        import seaborn as sns
```

Reading Dataset

Our main dataset is a collective aggregation of various tuples from multiple .mat datasets provided by NASA. The attributes of all the data are uniform. Removal of unnecessary labels is done later.

```
In [3]: dataset = pd.read_csv('Input n Capacity.csv')
        dataset = dataset.drop(labels=['SampleId'], axis=1)
```

The contesting attributes:

There are a total of 6 attributes that are taken into consideration for the mode. Out of these 6 attributes, one label is taken as a target value.

```
In [4]: print( 'Shape of DataFrame: {}'.format(dataset.shape))
        print (dataset.loc[1])

Shape of DataFrame: (636, 6)
Cycle                        0.000000
Time Measured(Sec)        3690.234000
Voltage Measured(V)          2.475768
Current Measured            -2.009436
Temperature Measured        39.162987
Capacity(Ah)                 2.035338
Name: 1, dtype: float64
```

```
In [5]: dataset.loc[280:]
```

Out[5]:

	Cycle	Time Measured(Sec)	Voltage Measured(V)	Current Measured	Temperature Measured	Capacity(Ah)
280	70	3148.829	3.484004	-0.000984	36.466500	1.622125
281	70	3148.829	3.466023	-0.001288	34.587894	1.530157
282	70	3148.829	3.238140	-0.000266	39.408643	1.667437
283	70	3045.187	3.414321	0.005711	34.697069	1.533426
284	71	3149.094	3.508463	-0.003644	36.017069	1.611326
...	...	...	...	...	...	...
631	166	2802.016	3.693059	-0.001245	32.371097	1.174975
632	166	2802.016	3.372148	-0.001992	38.470677	1.421787
633	167	2820.390	3.589937	-0.000583	34.405920	1.325079
634	167	2820.390	3.691809	-0.003127	32.192324	1.185675
635	167	2820.390	3.383857	-0.000985	37.851602	1.432455

356 rows × 6 columns

```
In [6]: data = dataset[~dataset.isin(['?'])]
        data.loc[280:]
```

Out[6]:

	Cycle	Time Measured(Sec)	Voltage Measured(V)	Current Measured	Temperature Measured	Capacity(Ah)
280	70	3148.829	3.484004	-0.000984	36.466500	1.622125
281	70	3148.829	3.468023	-0.001288	34.587894	1.530157
282	70	3148.829	3.238140	-0.000266	39.408643	1.667437
283	70	3045.187	3.414321	0.005711	34.697069	1.533426
284	71	3149.094	3.508463	-0.003644	36.017069	1.611326
...	...	...	...	...	...	...
631	166	2802.016	3.693059	-0.001245	32.371097	1.174975
632	166	2802.016	3.372148	-0.001992	38.470677	1.421787
633	167	2820.390	3.589937	-0.000583	34.405920	1.325079
634	167	2820.390	3.691809	-0.003127	32.192324	1.185675
635	167	2820.390	3.383857	-0.000985	37.851602	1.432455

356 rows × 6 columns

Dropping the rows/columns having null values:

```
In [8]: print(data.shape)
        print(data.dtypes)
```

```
(636, 6)
Cycle                        int64
Time Measured(Sec)         float64
Voltage Measured(V)        float64
Current Measured           float64
Temperature Measured       float64
Capacity(Ah)               float64
dtype: object
```

Finding data keys and data types for ANN model purpose:

```
In [9]: data.keys()
        data.shape
```

```
Out[9]: (636, 6)
```

```
In [10]: data = data.apply(pd.to_numeric)
         data.dtypes
```

```
Out[10]: Cycle                        int64
         Time Measured(Sec)         float64
         Voltage Measured(V)        float64
         Current Measured           float64
         Temperature Measured       float64
         Capacity(Ah)               float64
         dtype: object
```

Describing the data:

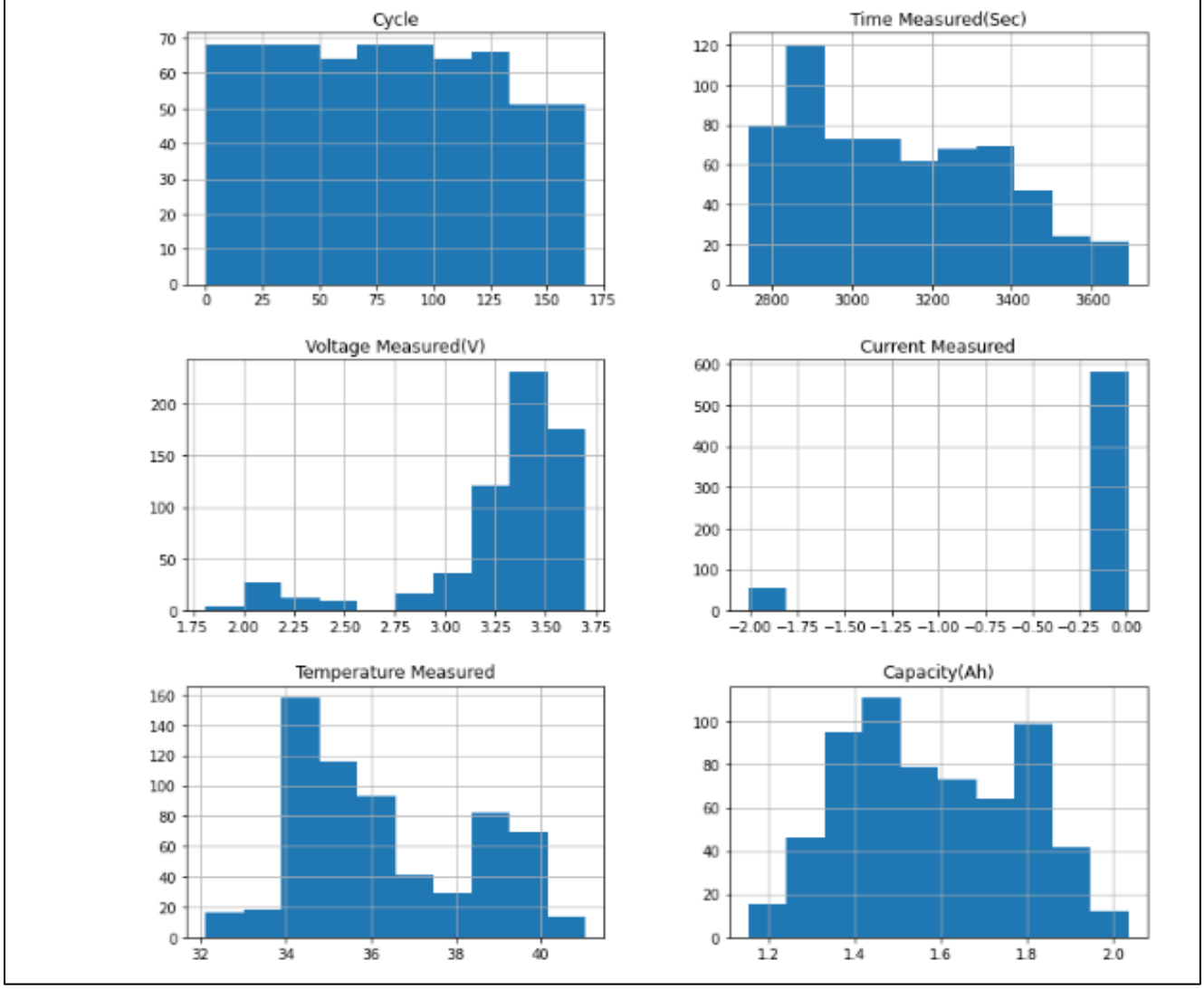
```
In [11]: data.describe()
```

Out[11]:

	Cycle	Time Measured(Sec)	Voltage Measured(V)	Current Measured	Temperature Measured	Capacity(Ah)
count	636.000000	636.000000	636.000000	636.000000	636.000000	636.000000
mean	79.764151	3116.977701	3.297086	-0.171153	36.318064	1.581652
std	47.137103	242.197224	0.382406	0.556974	2.090171	0.198765
min	0.000000	2742.643000	1.813269	-2.012015	32.113473	1.153818
25%	39.000000	2891.996250	3.260587	-0.003576	34.639503	1.421123
50%	79.000000	3084.281000	3.397571	-0.001903	35.808964	1.559695
75%	119.000000	3311.828000	3.529257	-0.000338	38.447301	1.763486
max	167.000000	3690.234000	3.697170	0.009113	41.049942	2.035338

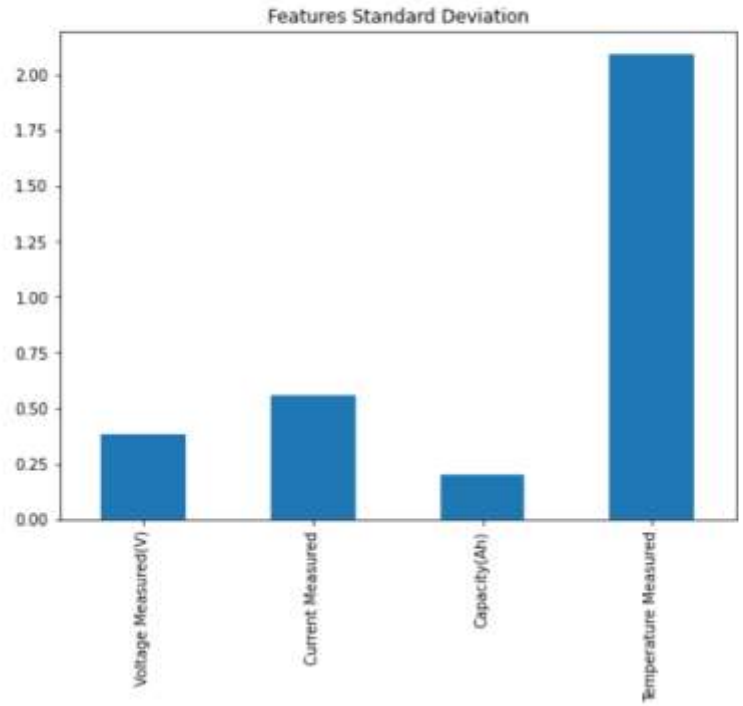
Plotting the graphs that showcase the range of values belonging to each attribute:-

```
In [12]: data.hist(figsize = (12, 12))
         plt.show()
```



Plotting graph for features and standard deviation:

```
In [13]: features=["Voltage Measured(V)","Current Measured","Capacity(Ah)","Temperature Measured"]
         data[features].std().plot(kind='bar', figsize=(8,6), title="Features Standard Deviation")

Out[13]: <AxesSubplot:title={'center':'Features Standard Deviation'}>
```



Splitting the main dataset:

X involves all the measurement attributes and y includes the values of the target attribute

```
In [14]: X = np.array(data.iloc[:,0:5].values)
         y = np.array(data.iloc[:,5].values)

In [15]: X

Out[15]: array([[ 0.00000000e+00,  3.69023400e+03,  3.27716998e+00,
                 -6.52835100e-03,  3.42308528e+01],
                [ 0.00000000e+00,  3.69023400e+03,  2.47576776e+00,
                 -2.00943589e+00,  3.91629865e+01],
                [ 0.00000000e+00,  3.69023400e+03,  3.06211271e+00,
                 -1.43329900e-03,  3.73384785e+01],
                ...,
                [ 1.67000000e+02,  2.82039000e+03,  3.58993739e+00,
                 -5.83347000e-04,  3.44059205e+01],
                [ 1.67000000e+02,  2.82039000e+03,  3.69180898e+00,
                 -3.12712900e-03,  3.21923241e+01],
                [ 1.67000000e+02,  2.82039000e+03,  3.38385665e+00,
                 -9.84538000e-04,  3.78516025e+01]])
```

```
In [16]: y

Out[16]: array([1.85648742, 2.03533759, 1.89105229, 1.85500452, 1.84632725,
                2.02514025, 1.88063703, 1.84319553, 1.83534919, 2.01332637,
                1.88066267, 1.83960184, 1.83526253, 2.01328467, 1.8807709 ,
                1.8306736 , 1.83464551, 2.00052834, 1.87945087, 1.83270021,
                1.83566166, 2.01389908, 1.88070035, 1.82852889, 1.83514614,
                2.01310111, 1.87993525, 1.82120119, 1.82575679, 1.96878983,
                1.88150881, 1.81517001, 1.82477385, 1.96816618, 1.86969079,
                1.80429805, 1.82461327, 1.95723079, 1.87005238, 1.82310023,
                1.82461955, 1.94559915, 1.87004424, 1.81212535, 1.81420194,
                1.9347505 , 1.8596519 , 1.80469164, 1.81375216, 1.92327995,
                1.85907466, 1.79084435, 1.81344049, 1.91188993, 1.85900846,
                1.78347072, 1.802598  , 1.90106671, 1.85936226, 1.78093861,
                1.8021069 , 1.889199  , 1.85873555, 1.77120904, 1.8025795 ,
                1.87827837, 1.84781729, 1.7686304 , 1.80306831, 1.86756972,
                1.84852529, 1.75363048, 1.80277763, 1.86758926, 1.84837895,
                1.74621974, 1.847026  , 1.97962655, 1.88078054, 1.73766473,
                1.84741731, 1.95754991, 1.88147216, 1.73151667, 1.83617742,
                1.94581593, 1.88100543, 1.79859409, 1.82578975, 1.82424611,
                1.47224801, 1.35470392, 1.30023574, 1.46701067, 1.349315  ,
                1.28950667, 1.46176725, 1.34418919, 1.27925306, 1.45669527,
                1.33899138, 1.27920782, 1.45135497, 1.33891452, 1.27403676,
                1.45161593, 1.33400668, 1.27412739, 1.44681613, 1.32864443,
                1.26376394, 1.44675706, 1.3231709 , 1.25813737, 1.44137975,
                1.31816916, 1.25325986, 1.43624562, 1.31846644, 1.25323133,
                1.43681542, 1.31829304, 1.2480875 , 1.43629451, 1.32387242,
                1.25343543, 1.44185753, 1.36012168, 1.28980488, 1.46720635,
                1.33953069, 1.2635402 , 1.45186404, 1.32902866, 1.24797617,
                1.44713797, 1.32367413, 1.23212029, 1.44179059, 1.3186339 ,
                1.22711377, 1.43163095, 1.31347513, 1.22173976, 1.43180843,
                1.31320206, 1.21110268, 1.42625626, 1.30779599, 1.20561598,
                1.42608775, 1.30303292, 1.20089363, 1.42126335, 1.30335736,
                1.1905904 , 1.41632727, 1.30341004, 1.18517925, 1.41657818,
                1.29788708, 1.17967068, 1.41084017, 1.29807351, 1.17459528,
                1.41099496, 1.29346361, 1.15381833, 1.40617143, 1.28800339,
                1.16440096, 1.40633585, 1.28745252, 1.15879749, 1.40045524,
                1.30901536, 1.17497488, 1.42178651, 1.32507933, 1.18567523,
                1.43245527])
```

Leveling the attributes' values:

As seen in X, the numerical values vary in great lengths. Hence it's necessary to bring these values to a common range. The mean and standard deviation of the data is calculated.

The mean is deducted from X.

The standard deviation is used to divide X.

```
In [17]: mean = X.mean(axis=0)
         X -= mean
         std = X.std(axis=0)
         X /= std

In [18]: mean
         std

Out[18]: array([ 47.10003109, 242.00674287,   0.38210518,   0.55653546,
                  2.08852755])
```

Creating X and y datasets for training:

```
In [19]: from sklearn import model_selection

         X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y, random_state=42, test_size = 0.3)

In [20]: X_test.shape

Out[20]: (191, 5)
```

Converting the y vectors (integers) to binary matrices.

```
In [21]: from keras.utils.np_utils import to_categorical

         Y_train = to_categorical(y_train, num_classes=None)
         Y_test = to_categorical(y_test, num_classes=None)
         print (Y_train.shape)
         print (Y_train[:10])

(445, 3)
[[0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]]
```

```
In [22]: X_train[0]
Out[22]: array([ 0.06870163, -0.12955301,  0.0497592 ,  0.30729039,  1.381131  ])
```

Creating a model:

Sequential provides training and inference features on this model.

```
In [23]: from keras.models import Sequential
         from keras.layers import Dense
         from tensorflow.keras.optimizers import Adam
         from keras.layers import Dropout
         from keras import regularizers

         # define a function to build the keras model
         def create_model():
             # create model
             model = Sequential()
             model.add(Dense(10, input_dim=5, kernel_initializer='uniform', activation='tanh'))
             model.add(Dropout(0.15))
             model.add(Dense(7, kernel_initializer='uniform', activation='relu'))
             model.add(Dropout(0.15))
             model.add(Dense(3,kernel_initializer='uniform', activation='sigmoid'))

             # compile model
             model.compile(loss='mean_squared_error', optimizer='sgd', metrics=['mean_absolute_error','accuracy'])
             return model

         model = create_model()

         print(model.summary())
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 10)                60

 dropout (Dropout)           (None, 10)                0

 dense_1 (Dense)             (None, 7)                 77

 dropout_1 (Dropout)         (None, 7)                 0

 dense_2 (Dense)             (None, 3)                 24

=================================================================
Total params: 161
Trainable params: 161
Non-trainable params: 0
_________________________________________________________________
None
```

Fitting the proposed model:

```
In [24]: history=model.fit(X_train, Y_train, validation_data=(X_test, Y_test),epochs=10, batch_size=1)

Epoch 1/10
445/445 [==============================] - 11s 6ms/step - loss: 0.2107 - mean_absolute_error: 0.4583 - accuracy: 0.9865 - val_l
oss: 0.1754 - val_mean_absolute_error: 0.4187 - val_accuracy: 0.9948
Epoch 2/10
445/445 [==============================] - 2s 5ms/step - loss: 0.1508 - mean_absolute_error: 0.3873 - accuracy: 0.9865 - val_lo
ss: 0.1272 - val_mean_absolute_error: 0.3562 - val_accuracy: 0.9948
Epoch 3/10
445/445 [==============================] - 2s 4ms/step - loss: 0.1125 - mean_absolute_error: 0.3335 - accuracy: 0.9865 - val_lo
ss: 0.0962 - val_mean_absolute_error: 0.3093 - val_accuracy: 0.9948
Epoch 4/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0875 - mean_absolute_error: 0.2930 - accuracy: 0.9865 - val_lo
ss: 0.0755 - val_mean_absolute_error: 0.2735 - val_accuracy: 0.9948
Epoch 5/10
445/445 [==============================] - 2s 3ms/step - loss: 0.0706 - mean_absolute_error: 0.2617 - accuracy: 0.9865 - val_lo
ss: 0.0613 - val_mean_absolute_error: 0.2457 - val_accuracy: 0.9948
Epoch 6/10
445/445 [==============================] - 2s 3ms/step - loss: 0.0588 - mean_absolute_error: 0.2370 - accuracy: 0.9865 - val_lo
ss: 0.0510 - val_mean_absolute_error: 0.2234 - val_accuracy: 0.9948
Epoch 7/10
445/445 [==============================] - 2s 3ms/step - loss: 0.0501 - mean_absolute_error: 0.2173 - accuracy: 0.9865 - val_lo
ss: 0.0433 - val_mean_absolute_error: 0.2051 - val_accuracy: 0.9948
Epoch 8/10
445/445 [==============================] - 2s 3ms/step - loss: 0.0435 - mean_absolute_error: 0.2006 - accuracy: 0.9865 - val_lo
ss: 0.0374 - val_mean_absolute_error: 0.1898 - val_accuracy: 0.9948
Epoch 9/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0385 - mean_absolute_error: 0.1867 - accuracy: 0.9865 - val_lo
ss: 0.0327 - val_mean_absolute_error: 0.1768 - val_accuracy: 0.9948
Epoch 10/10
445/445 [==============================] - 2s 4ms/step - loss: 0.0345 - mean_absolute_error: 0.1749 - accuracy: 0.9865 - val_lo
ss: 0.0290 - val_mean_absolute_error: 0.1656 - val_accuracy: 0.9948
```

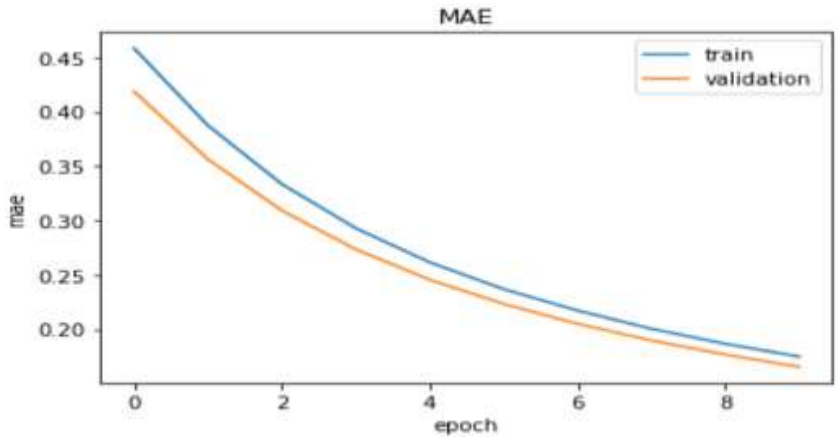
```
In [25]: history.history

Out[25]: {'loss': [0.2106662094593048,
           0.15080635249614716,
           0.11245765537023544,
           0.08751964569091797,
           0.07064806669950485,
           0.0587754100561142,
           0.05014600604772568,
           0.04354841634631157,
           0.03850608319044113,
           0.034466881304979324],
          'mean_absolute_error': [0.458296000957489,
           0.38730376958847046,
           0.3335109055042267,
           0.2929958999156952,
           0.2616851329803467,
           0.236993208527565,
           0.21729083359241486,
           0.20055419206619263,
           0.1867244392633438,
           0.17491520941257477],
          'accuracy': [0.9865168333053589,
           0.9865168333053589,
           0.9865168333053589,
           0.9865168333053589,
           0.9865168333053589,
           0.9865168333053589,
```

```
'accuracy': [0.9865168333053589,
 0.9865168333053589,
 0.9865168333053589,
 0.9865168333053589,
 0.9865168333053589,
 0.9865168333053589,
 0.9865168333053589,
 0.9865168333053589,
 0.9865168333053589],
'val_loss': [0.17540042102336884,
 0.12718963623046875,
 0.09615728259086609,
 0.0755392387509346,
 0.06126789376139641,
 0.050982631742954254,
 0.04330302029848099,
 0.0373942069709301,
 0.032739005982875824,
 0.028983628377318382],
'val_mean_absolute_error': [0.41869622468948364,
 0.35622578859329224,
 0.3092613220214844,
 0.2735223174095154,
 0.24567191302776337,
 0.22339007258415222,
 0.20512676239013672,
 0.18983782827854156,
 0.17682671546936035,
 0.16555511951446533],
```

```
'val_accuracy': [0.9947643876075745,
 0.9947643876075745,
 0.9947643876075745,
 0.9947643876075745,
 0.9947643876075745,
 0.9947643876075745,
 0.9947643876075745,
 0.9947643876075745,
 0.9947643876075745]}
```

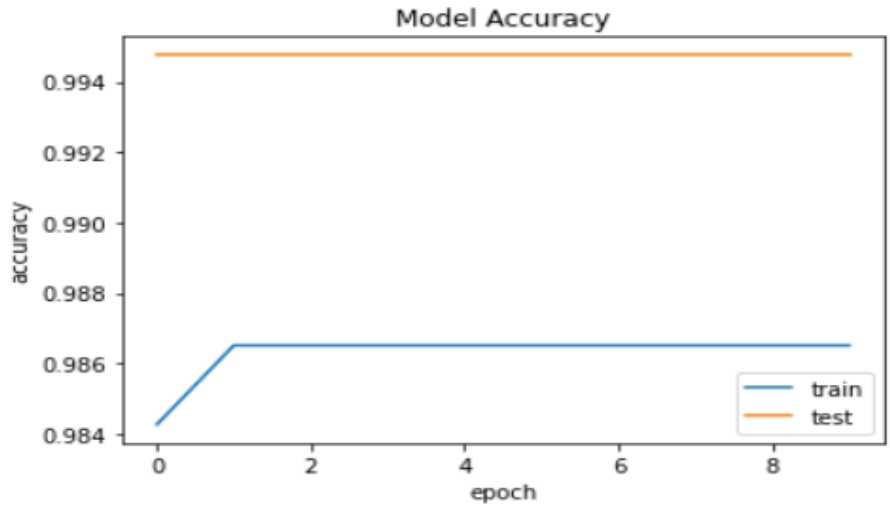
Plotting the obtained and predicted Mean Absolute Error (MAE):

```
In [26]: plt.plot(history.history['mean_absolute_error'])
         plt.plot(history.history['val_mean_absolute_error'])
         plt.title('MAE')
         plt.ylabel('mae')
         plt.xlabel('epoch')
         plt.legend(['train', 'validation'], loc='upper right')
         plt.show()
```



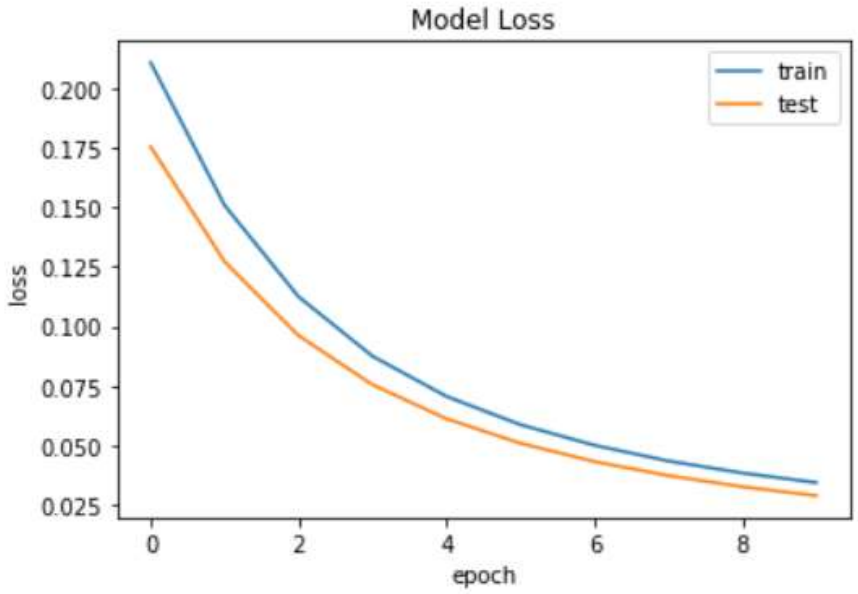
Plotting the obtained and predicted Accuracy:

```
In [28]: import matplotlib.pyplot as plt
         %matplotlib inline
         # Model accuracy
         plt.plot(history.history['accuracy'])
         plt.plot(history.history['val_accuracy'])
         plt.title('Model Accuracy')
         plt.ylabel('accuracy')
         plt.xlabel('epoch')
         plt.legend(['train', 'test'])
         plt.show()
```



Plotting the obtained and predicted Mean Squared Error (MSE):

```
In [28]: plt.plot(history.history['loss'])
         plt.plot(history.history['val_loss'])
         plt.title('Model Loss')
         plt.ylabel('loss')
         plt.xlabel('epoch')
         plt.legend(['train', 'test'])
         plt.show()
```



Plotting the graph of Cycle Vs Capacity

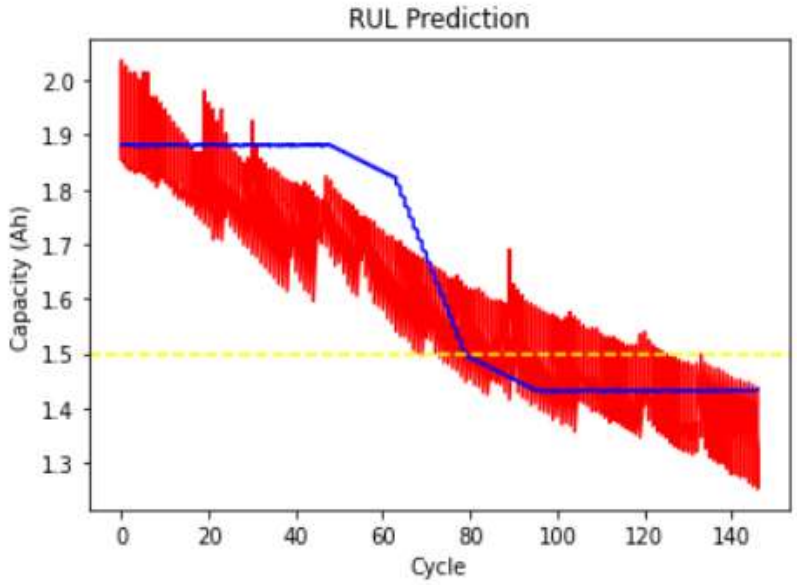
Graphs and Inferences: RUL Prediction using SGD optimizer:

Details of Neural Network :

- Input Layer: 1 neuron, Activation function= tanh
- Two Hidden layers: Sigmoid and RELU are hidden layers, along with 2 other dense functions.
- Input to the neural network : Cycle No.
- Output of the network : Capacity
- Optimizer : SGD Loss: Mean squared error

The Red line indicates capacity values for the battery dataset. The Blue line indicates the predicted values using NN (optimizer=sgd). The Neural network was trained on Batteries B0006, B0007, B0018 from the dataset provided by NASA However it is observed from the trails performed during implementation that prediction based on only cycle number does give accurate results.

```
y_prediction = [*f, *e, *d]
import matplotlib.pyplot as plt
threshold=1.5
#X_Cycle = X_test[:,0]
X_Cycle = np.array(data.iloc[:573,0].values)
Y_Capacity = np.array(data.iloc[:573,5].values)
plt.plot(X_Cycle, Y_Capacity, color='red')
plt.plot(X_Cycle, y_prediction, color='blue')
plt.axhline(threshold, color='yellow', linestyle='--')
plt.title('RUL Prediction')
plt.xlabel('Cycle')
plt.ylabel('Capacity (Ah)')
plt.show()
```

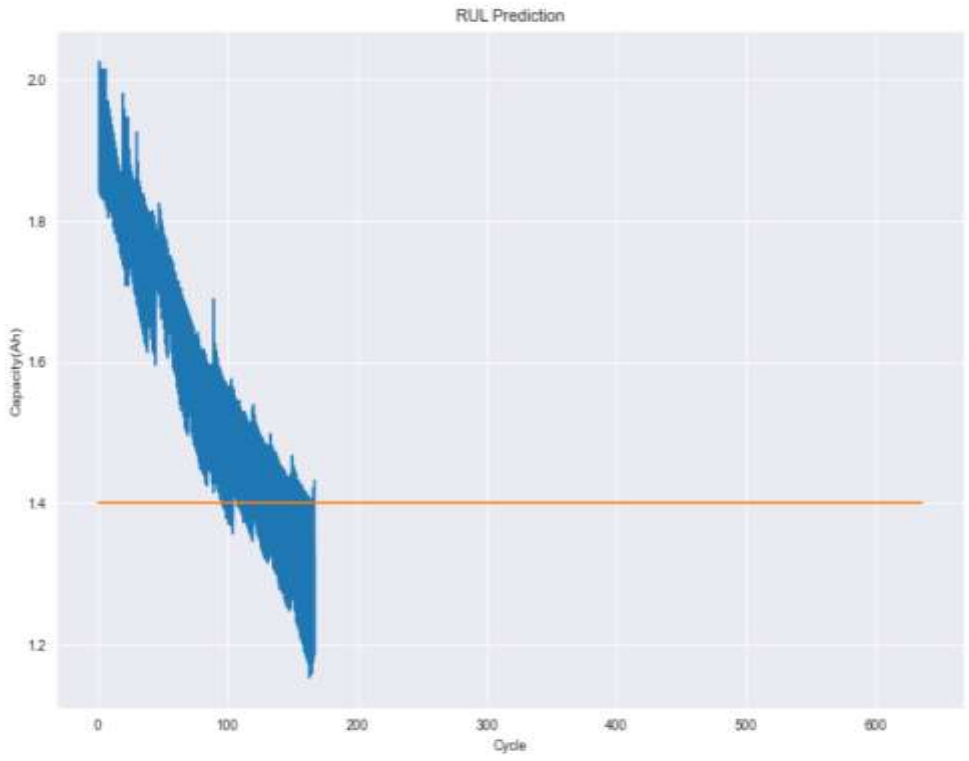


Following Graphs Of Cycle Vs Capacity Is Plotted For Analyzing Data:

```
In [30]: %matplotlib inline
         import matplotlib.pyplot as plt
         import seaborn as sns
         plot_df = data.loc[(data['Cycle']>=1),['Cycle','Capacity(Ah)']]
         sns.set_style("darkgrid")
         plt.figure(figsize=(12, 8))
         plt.plot(plot_df['Cycle'], plot_df['Capacity(Ah)'])
         #Draw threshold
         plt.plot([0.,len(data)], [1.4, 1.4])
         plt.ylabel('Capacity(Ah)')
         # make x-axis ticks legible
         adf = plt.gca().get_xaxis().get_major_formatter()
         plt.xlabel('Cycle')
         plt.title('RUL Prediction')
```

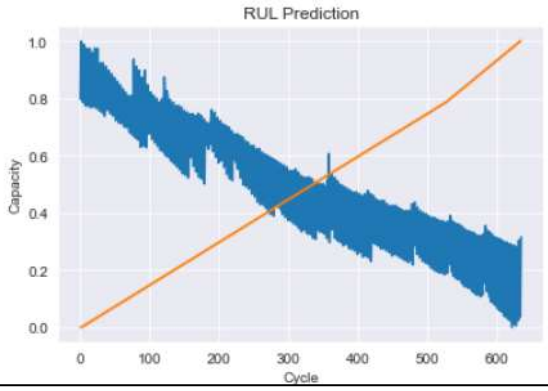
This graph is based off of the real untouched values of the dataset.

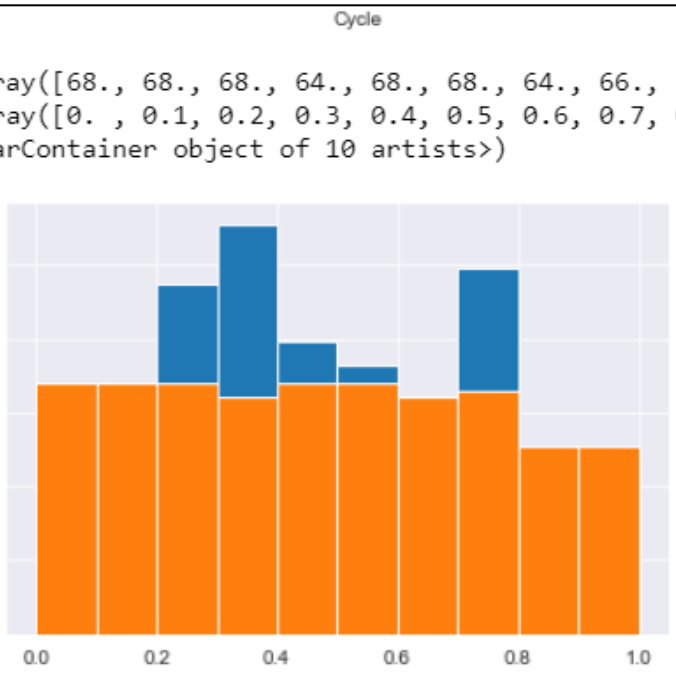
```
Out[30]: Text(0.5, 1.0, 'RUL Prediction')
```



In this graph , the values are normalized, so that both attributes stand on the same level .

```
In [31]: capacity = np.array(data.iloc[:,5].values)
         cycle = np.array(data.iloc[:,0].values)
         normalizedCycle = (cycle-min(cycle))/(max(cycle)-min(cycle))
         normalizedCapacity = (capacity-min(capacity)) / (max(capacity) - min(capacity))
         plt.plot(normalizedCapacity)
         plt.plot(normalizedCycle)
         plt.title('RUL Prediction')
         plt.ylabel('Capacity')
         plt.xlabel('Cycle')
         plt.show()
         plt.hist(normalizedCapacity, bins=10)
         plt.hist(normalizedCycle, bins=10)
```



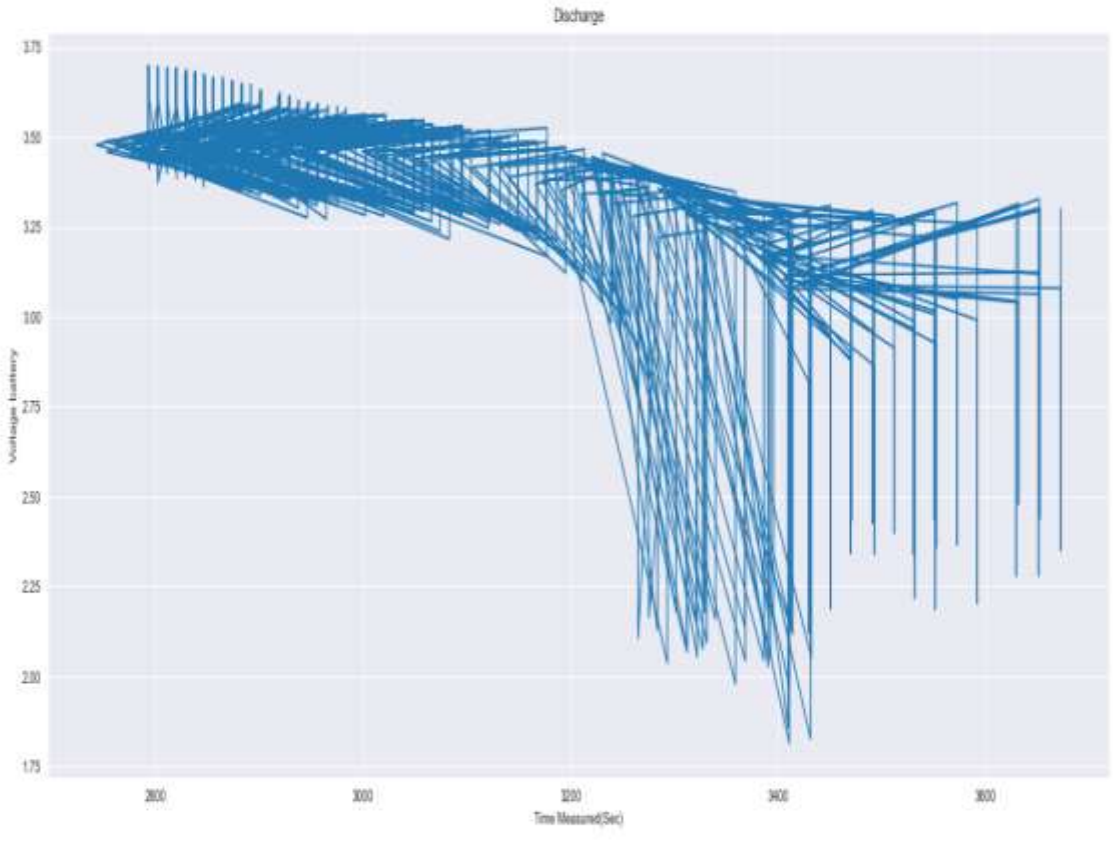



```
Out[31]: (array([68., 68., 68., 64., 68., 68., 64., 66., 51., 51.]),
          array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ]),
          <BarContainer object of 10 artists>)
```

Plotting the graph of Time Measured Vs Voltage Measured for all cycles:-

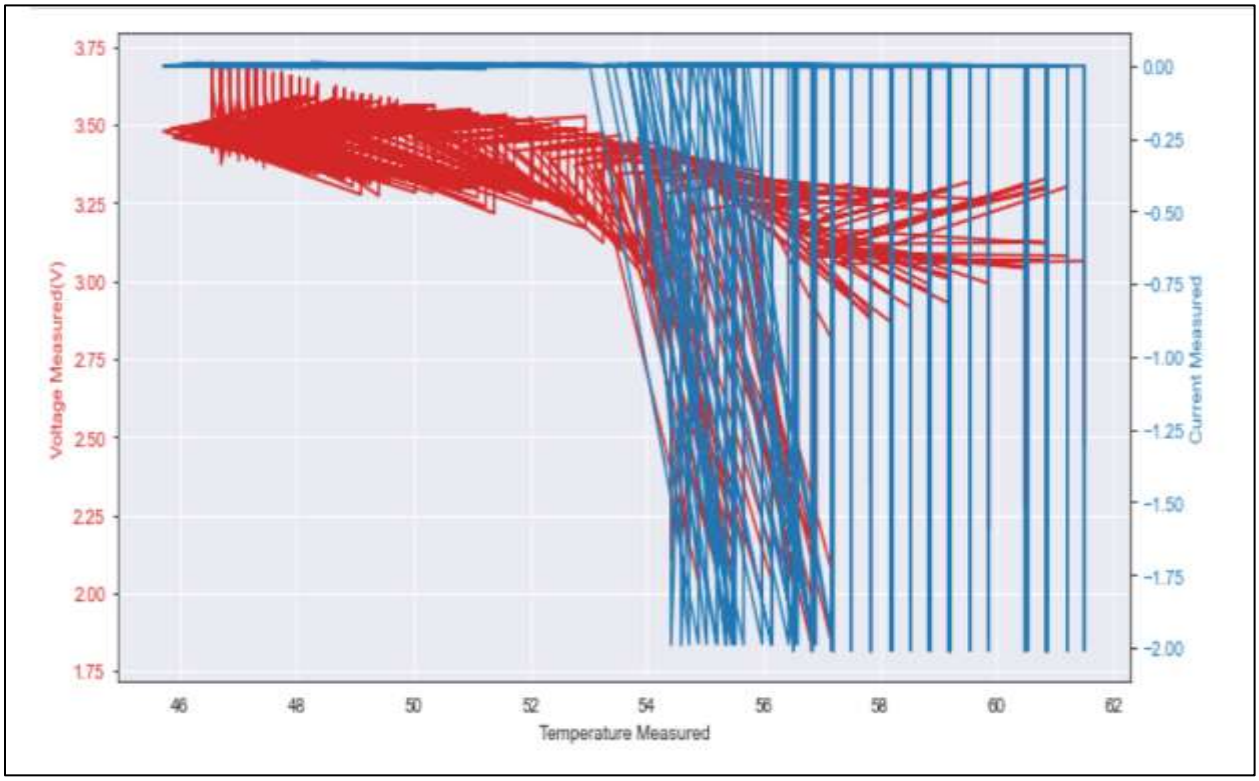
```
In [32]: %matplotlib inline
         import matplotlib.pyplot as plt
         import seaborn as sns

         plot_df = data.loc[(data['Cycle']>=1),['Time Measured(Sec)','Voltage Measured(V)']]
         sns.set_style("darkgrid")
         plt.figure(figsize=(20, 8))
         plt.plot(plot_df['Time Measured(Sec)'], plot_df['Voltage Measured(V)'])
         plt.ylabel('Voltage battery')

         # make x-axis ticks legible
         adf = plt.gca().get_xaxis().get_major_formatter()
         #adf.scaled[1.0] = '%m-%d-%Y'
         plt.xlabel('Time Measured(Sec)')
         plt.title('Discharge')
```

```
Out[32]: Text(0.5, 1.0, 'Discharge')
```



Plotting the observed relationship between Temperature Measured, Voltage Measured, and Current Measured

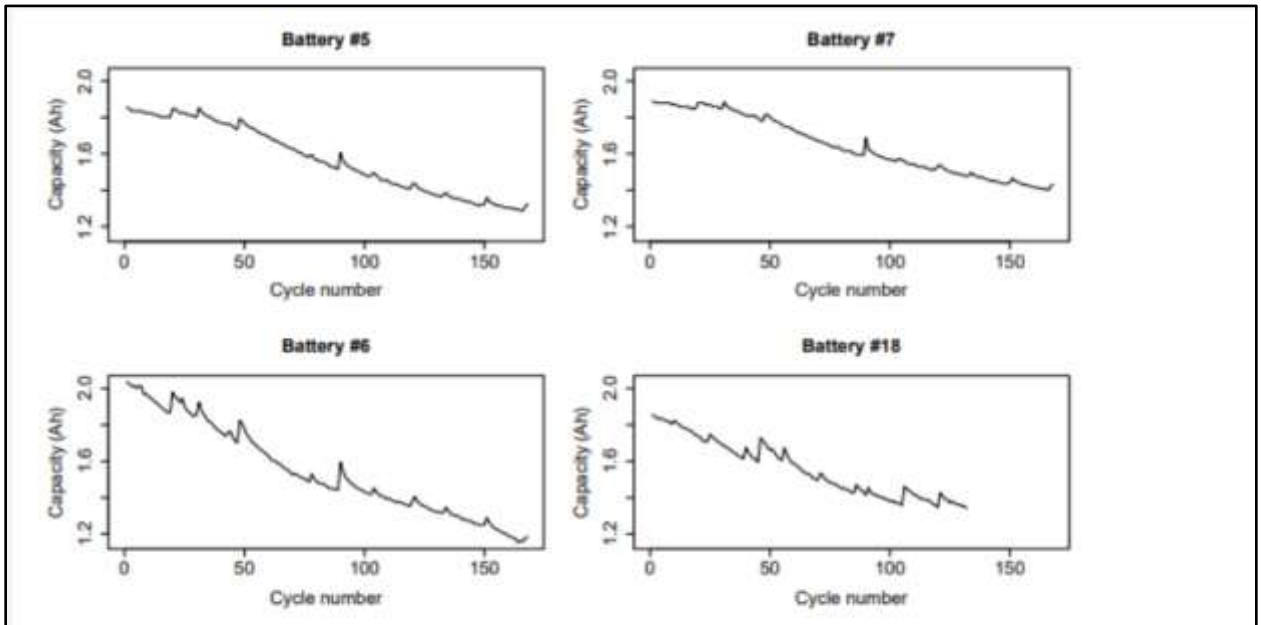
```
In [33]: import numpy as np
         import matplotlib.pyplot as plt
         %matplotlib inline
         # Create some mock data

         fig, ax1 = plt.subplots()
         sns.set_style("white")
         plot_df= data.loc[(data['Cycle']==1),['Temperature Measured','Current Measured']]
         #plt.plot([126, 127], color="black")
         plot_df1 = data.loc[(data['Cycle']==1),['Temperature Measured','Voltage Measured(V)']]

         color = 'tab:red'
         ax1.set_xlabel('Temperature Measured')
         ax1.set_ylabel('Voltage Measured(V)', color=color)
         ax1.plot(data['Time Measured(Sec)']/60, data['Voltage Measured(V)'], color=color)
         ax1.tick_params(axis='y', labelcolor=color)

         ax2 = ax1.twinx()  # instantiate a second axes that shares the same x-axis

         color = 'tab:blue'
         ax2.set_ylabel('Current Measured', color=color)  # we already handled the x-label with ax1
         ax2.plot(data['Time Measured(Sec)']/60, data['Current Measured'],'-', color=color)
         ax2.tick_params(axis='y', labelcolor=color)
         fig.set_size_inches(10, 5)
         fig.tight_layout()  # otherwise the right y-label is slightly clipped
         plt.show()
         # vertical black line split the graph between charge and discharge operation.
```



Discussion

Introduction to NASA Dataset:

There are four Li-ion batteries (#5, #6, #7, and #18), which were run through 3 different operational profiles (charge, discharge, and impedance) at room temperature. Charging was carried out in constant current (CC) mode at 1.5 A until the battery voltage reached 4.2 V and then continued in a constant voltage (CV) mode until the charge current dropped to 20 mA. Discharge was carried out at a constant current (CC) level of 2 A until the battery voltage fell to 2.7 V, 2.5 V, 2.2 V, and 2.5 V for batteries #5, #6, #7, and #18, respectively. The experiments were stopped when the batteries reached the EOL criterion, which was a 30% fade in rated capacity (from 2 Ahr to 1.4 Ahr). Figure 4 shows the relationship between the capacity and cycle number. The number of cycles are 168, 168, 168, 132, for batteries #5, #6, #7, and #18.



Plots of the data: battery capacity based on the cycle number.

Sample of NASA Dataset:

Type	Time_Start	Time_End	Capacity	v_1	...	i_1	...	t_1	...
charge	2008/4/2 13:08	2008/4/2 15:14		4.187		0.643		25.32	
discharge	2008/4/2 15:25	2008/4/2 16:27	1.856487	3.530		−1.819		32.57	
charge	2008/4/2 16:37	2008/4/2 19:33		4.059		0.949		26.64	
discharge	2008/4/2 19:43	2008/4/2 20:45	1.846327	3.537		−1.818		32.73	
charge	2008/4/2 20:55	2008/4/2 23:50		4.058		0.951		26.78	
discharge	2008/4/3 0:01	2008/4/3 1:01	1.835349	3.544		−1.816		32.64	

In this project, we presented a modified ANN method based on various activation functions, which can be used to deal with condition monitoring data collected at discrete inspection time points.

The main reason for using ANN Model:

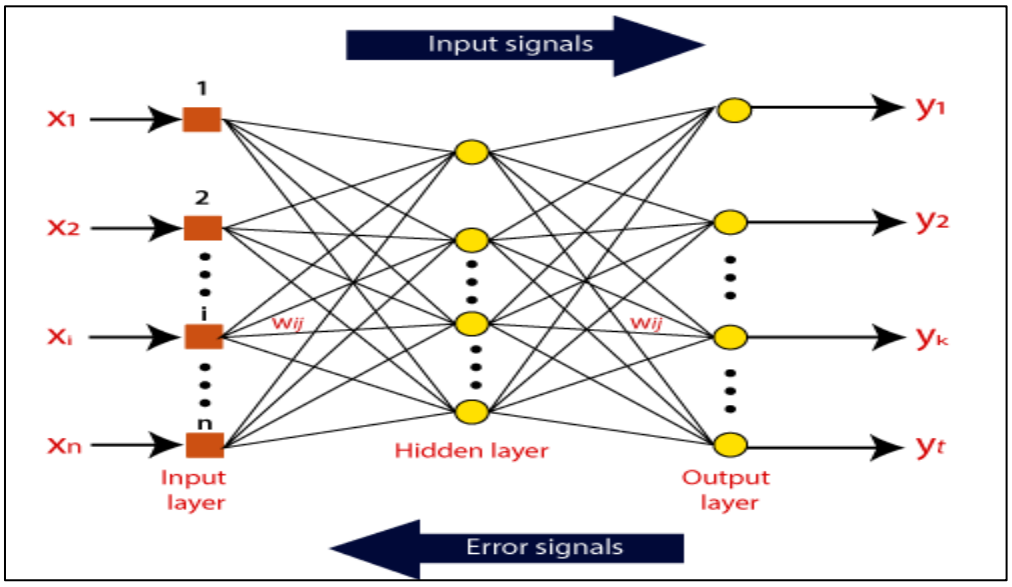
With the data's Capacity as target, we take into account the cycle and measurement values as well as the changes of the measurement values at hidden layers' points, and this important information will be processed by ANN to estimate the remaining useful life. We only use particular data pertaining to 5 attributes from each mat file instead of incorporating data in ever mat file, because ANNs with less input nodes have better generalization capability, and the experiments results in this work show that adding more input nodes will not improve the RUL prediction performance.

It is worth justifying the use of the remaining life to represent the health condition of a specified battery. However, it is true that the health condition of a battery deteriorates with time as well as discharge cycles, and, without loss of generality, we can assume that the true inherent health condition index increases monotonically with time and cycles. Since it is hard to determine the true inherent health condition index based on the collected condition monitoring measurements, we can try to identify a measure that has a monotonic mapping relationship with the true inherent health condition index. We believe "remaining life percentage" is a good option for this purpose: (1) the mapping between the inherent health condition index and the life percentage is monotonically non-decreasing, and (2) remaining life is also able to indicate when the degradation occurs. The remaining life

prediction problem is a complex nonlinear problem. The objective is to predict the battery's remaining useful life based on the available metrics and condition monitoring data. The relationship between the 5 inputs, and the output, the life percentage, is very complex and nonlinear. Because of the capability in modeling complex non-linear relationships, ANN is a powerful tool to be used to address the remaining useful life prediction problem in the proposed approach.

Working of artificial neural networks:

An Artificial Neural Network can be best represented as a weighted directed graph, where the artificial neurons form the nodes. The association between the neuron outputs and neuron inputs can be viewed as the directed edges with weights. The Artificial Neural Network receives the input signal from the external source in the form of a pattern and image in the form of a vector. These inputs are then mathematically assigned by the notations x(n) for every n number of inputs.



Afterward, each of the inputs is multiplied by its corresponding weights (these weights are the details utilized by the artificial neural networks to solve a specific problem). In general terms, these weights normally represent the strength of the interconnection between

neurons inside the artificial neural network. All the weighted inputs are summarized inside the computing unit.

If the weighted sum is equal to zero, then bias is added to make the output non-zero or something else to scale up to the system's response. Bias has the same input, and weight equals to 1. Here the total of weighted inputs can be in the range of 0 to positive infinity. Here, to keep the response in the limits of the desired value, a certain maximum value is benchmarked, and the total of weighted inputs is passed through the activation function.

The activation function refers to the set of transfer functions used to achieve the desired output. There is a different kind of activation function, but primarily either linear or non-linear sets of functions. Some of the commonly used sets of activation functions are the Binary, linear, and Tan hyperbolic sigmoidal activation functions.

Activation Functions used:

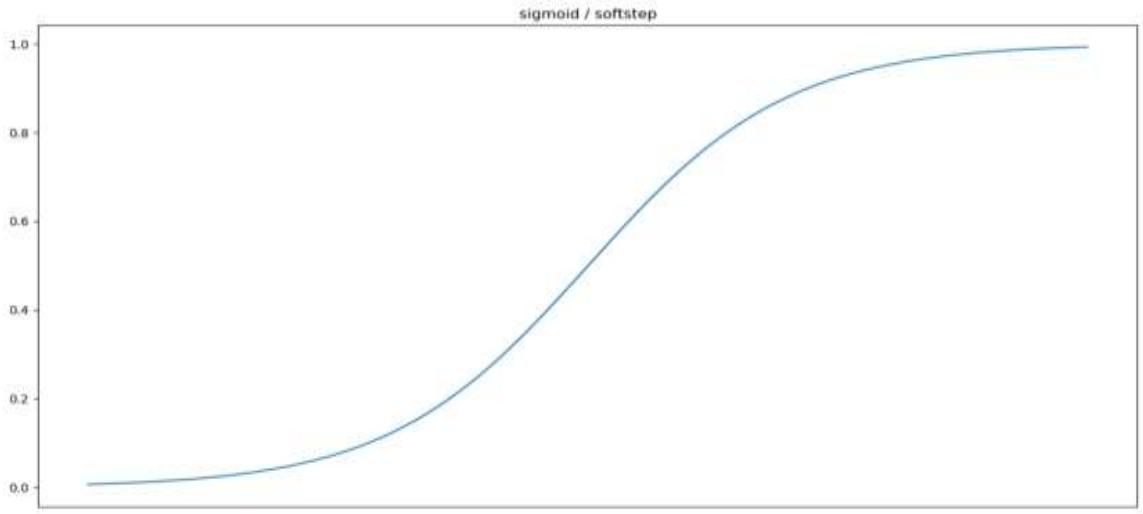
In any network, every neuron is composed of a weights vector and a bias value. When a new vector is input, it computes the dot product between the weights and the input vector, adds the bias value and outputs the scalar value. However in many cases, it does not. Because put very simply: both the dot product and the scalar additions are linear operations. Hence, when we have this value as neuron output and do this for every neuron, we have a system that behaves linearly. Keeping in mind, most data that we possess is highly nonlinear. Since linear neural networks would not be capable of generating a decision boundary in our case, there would be no point in applying them when generating predictive models. The system as a whole must therefore be nonlinear.

The function, which is placed directly behind every neuron, takes as input the linear neuron output and generates a nonlinear output based on it, often deterministically. In such a way, with every neuron generating in effect a linear-but-nonlinear output, the system behaves nonlinearly as well and by consequence becomes capable of handling our data. Activation outputs increase with input.

Our main activation function used in the model is ReLU. Activation functions in general are used to convert linear outputs of a neuron into nonlinear outputs, ensuring that a neural network can learn nonlinear behavior. Rectified Linear Unit (ReLU) does so by outputting x for all x >= 0 and 0 for all x < 0. In other words, it equals max(x, 0). This simplicity makes it more difficult than the Sigmoid activation function and the Tangens hyperbolicus (Tanh) activation function, which use more difficult formulas and are computationally more expensive. In addition, ReLU is not sensitive to vanishing gradients, whereas the other two are, slowing down learning in the network.

1. Sigmoid

In the following figure, a sigmoid function is presented, also known as the logistic curve:



Mathematically, it can be represented as follows:

$$f(x) = \frac{1}{1 + e^{-x}}$$

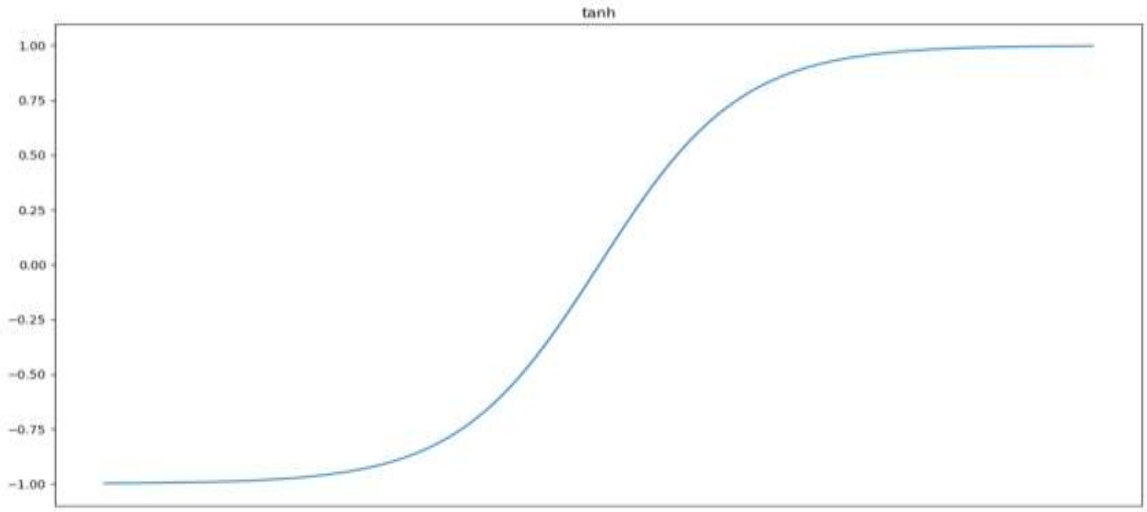
As seen in the plot, the function slowly increases over time, but the greatest increase can be found around x = 0. The range of the function is (0, 1); that is towards high values for x the function therefore approaches 1, but never equals it.

The Sigmoid function's step functions used in ancient neurons are not differentiable and hence gradient descent for optimization cannot be applied. Second, when we implemented the Rosenblatt perceptron, we noticed that in a binary classification problem, the decision boundary is optimized per neuron and will find one of the possible boundaries if they exist. This gets easier with the Sigmoid function, since it is more smooth.

Additionally, and perhaps primarily, we use the Sigmoid function because it outputs between (0, 1). When estimating a probability such as remaining life of Lithium Ion batteries, this is perfect, because probabilities have a very similar range of [0, 1]. Sigmoid functions allow us to give a very weighted estimate.

2. Tangens hyperbolicus: Tanh

Another applied activation function is the tangens hyperbolicus, or hyperbolic tangent / tanh function:



Mathematically, it can be represented as follows:

$$f(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$$

It works similar to the Sigmoid function, but is implemented regarding some differences.

First, the change in output accelerates close to x = 0, which is similar to the Sigmoid function.

It does also share its asymptotic properties with Sigmoid: although for very large values of x the function approaches 1, it never actually equals it.

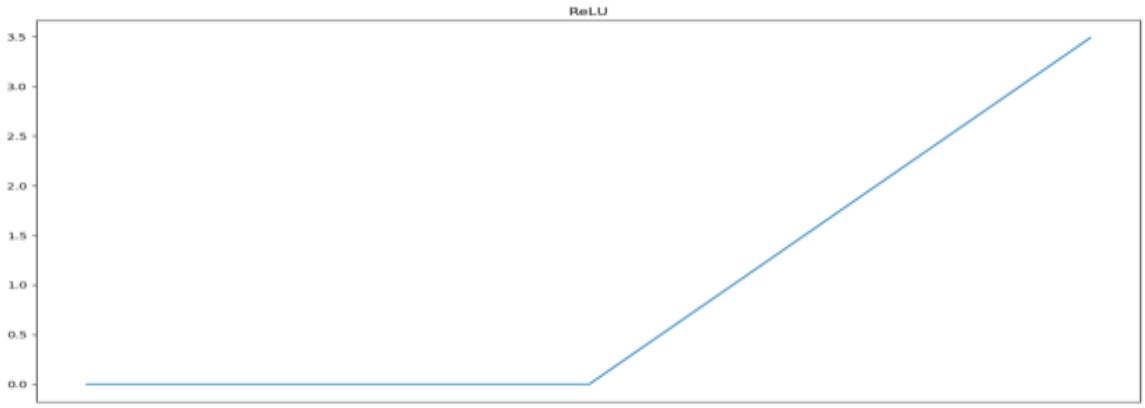
The range of the activation function differs: (-1, 1).

Although this difference seems to be very small, it showcases a large effect on the model performance. This is related to the fact that they are symmetric around the origin. Hence, they produce outputs that are close to zero. Outputs close to zero are preferable as we initially deduct our X value prior to optimization, they produce the least weight swings, and hence let our model converge faster.

3. Rectified Linear Unit: ReLU

In order to improve on our observations after applying tanh and sigmoid, another activation was used. Rectified Linear Unit or ReLU is much less sensitive to the problems faced in above activation functions and hence improved our training process.

It looks as follows:



And can be represented as follows:

$$f(x) = max\ (0, x)$$

Or, in layman terms, it produces a zero output for all inputs smaller than zero; and x for all other inputs. Hence, for all inputs <= 0, it produces zero outputs.

- Sparsity: This benefits sparsity substantially: in almost half the cases, now, the neuron doesn't fire anymore.
- Fewer vanishing gradients: It also reduces the impact of vanishing gradients, because the gradient is always a constant: the derivative of f(x) = 0 is 0 while the derivative of f(x) = x is 1. Model hence learns faster and more evenly.
- Computational requirements: Additionally, ReLU does need much fewer computational resources than the Sigmoid and Tanh functions. The function that essentially needs to be executed to arrive at ReLU is a max function: max(0, x) produces 0 when x < 0 and x when x >= 0.

Training of proposed ANN Model:

A critical issue of using ANN is to avoid overfitting the network. If an ANN is overfitted, noise factors will be modeled in the network, which affects the generalization capability of ANN, and thus affects the prediction accuracy. A widely used approach for avoiding overfitting is the use of a validation set. That is, during the ANN training process, the mean square error for the training set and that for the validation set are calculated. Both of the mean square errors drop early in the training process because the ANN is learning the relationship between the inputs and the outputs by modifying the trainable weights based on the training set. After a certain point, the mean square error for the validation set will start to increase, because the ANN starts to model the noise in the training set. Thus, the training process can be stopped at this point, and the trained ANN with good modeling and generalization capability can be achieved. Another question is how to construct the validation set. One method is dividing the available failure histories into two groups, one used as the training set and the other as the validation set. The percentage of histories used

in the validation set is typically around 40%. This causes a problem if we do not have a lot of failure histories, which is the case in many practical applications.

In this paper, we propose to construct the validation set using the actual measurement data from the collective dataset, and use the remaining fitted measurement data to construct the training set. In this way, data based on all attributes can be taken advantage of for modifying the trainable weights in the ANN training process, and the validation process can help to avoid overfitting the network. Experiments based on practical condition monitoring data have shown that the use of such a validation set in the training process can lead to more accurate and robust predictions.

Because of the uncertainty in the ANN training algorithms such as the LM algorithm, with the same training set and validation set, typically we will not obtain the same neural network after training. In this work, we train the ANN to obtain the lowest training mean square error (MSE). This is actually another advantage of using the validation mechanism in the training process. Without the validation process, a lower training MSE does not necessarily mean a better network, and in many cases it is not. Thus, we cannot select the best trained ANN based on the training MSE if we do not have the validation mechanism. However, if we do have the validation mechanism, as in the proposed ANN method, a lower training MSE will indicate a trained ANN with better modeling and generalization capabilities, and thus an ANN with better prediction capability.

Procedure of the proposed ANN method:

The explanation of the procedure is given as follows. We start from the available historical data, which includes the numerical values and actual condition monitoring measurement values.

- **Step 1**: Each measurement data for a battery is fitted in a collective dataset. The capacity values and the fitted measurement values of various mat datasets are used to construct the ANN training set.

- **Step 2**: The ANN validation set is constructed using the collective dataset and the actual measurement values as inspection points are the Capacity values.
- **Step 3**: Train the ANN model based on the training set and the validation set using the sequential model. This Step 3 gives the trained ANN model desired obtained for RUL prediction.
- **Step 4**: At every iteration, we first fit each measurement series based on the measurement values up to the current range. The fitted measurement values as well as the target values in these two sets, are used as the inputs to the trained ANN model.
- **Step 5**: The predicted remaining useful life from the current set of values is calculated using the trained ANN model.
- **Step 6**: The RUL is calculated based on the current capacity of the battery and the measured attributes.
- **Step 7**: For every epoch, repeat Step 4 to Step 6 based on the available data, and determine the RUL prediction.

RUL prediction using the proposed ANN method:

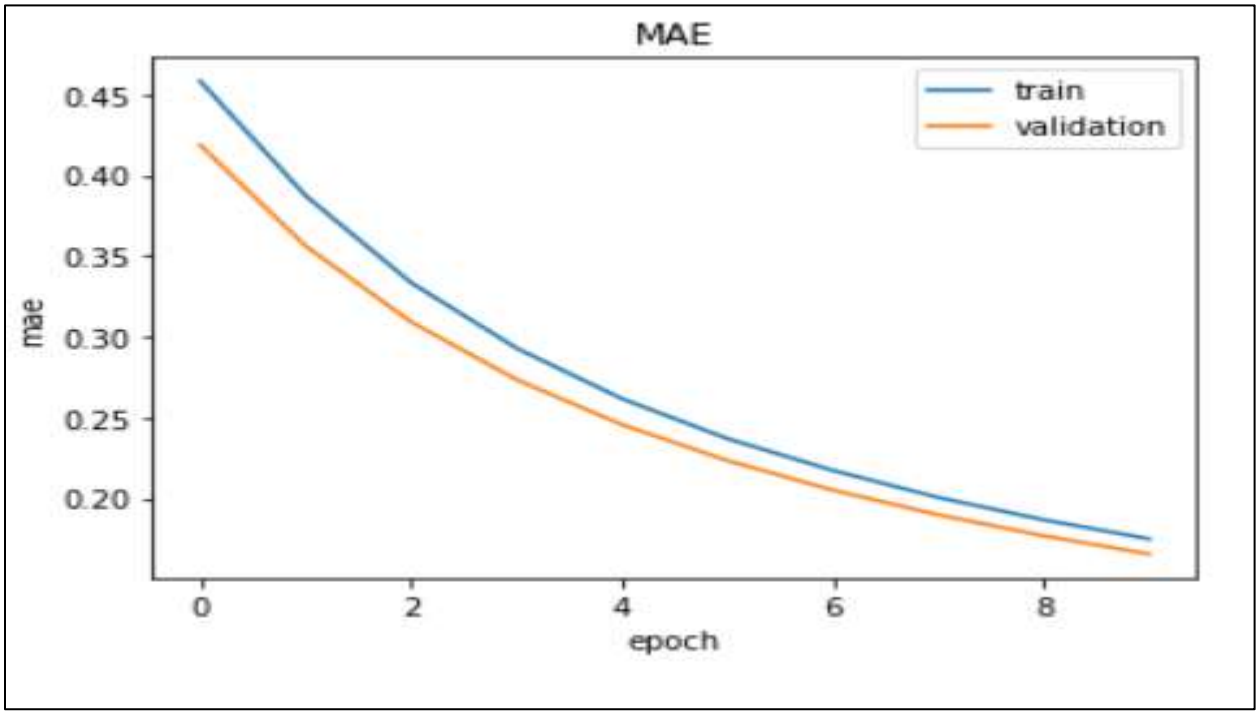
The collective data from NASA mentioned above are used to test the RUL prediction performance of the proposed batteries subjected to condition monitoring regarding various aspects in the ANN method. The following approach is used to test the RUL prediction performance of the proposed ANN method. When we apply the ANN prediction method in practical situations, we first use the available data, to train the ANN, and then use the trained ANN model to predict the RUL of a specific observation in the test dataset which then compares the capacity against the prediction. Thus, we use X_test and y_test sets to construct the ANN validation set to test the prediction performance, and use the remaining data to construct the ANN training set and the validation set to train the ANN model. After that, we can use a different slice approach to construct the ANN test set to test the prediction performance and use the remaining data to train the ANN. We can repeat this process until we go through all the well performed approaches and select a prime range. Using the approach described above, the test data is not involved in the ANN training process, and

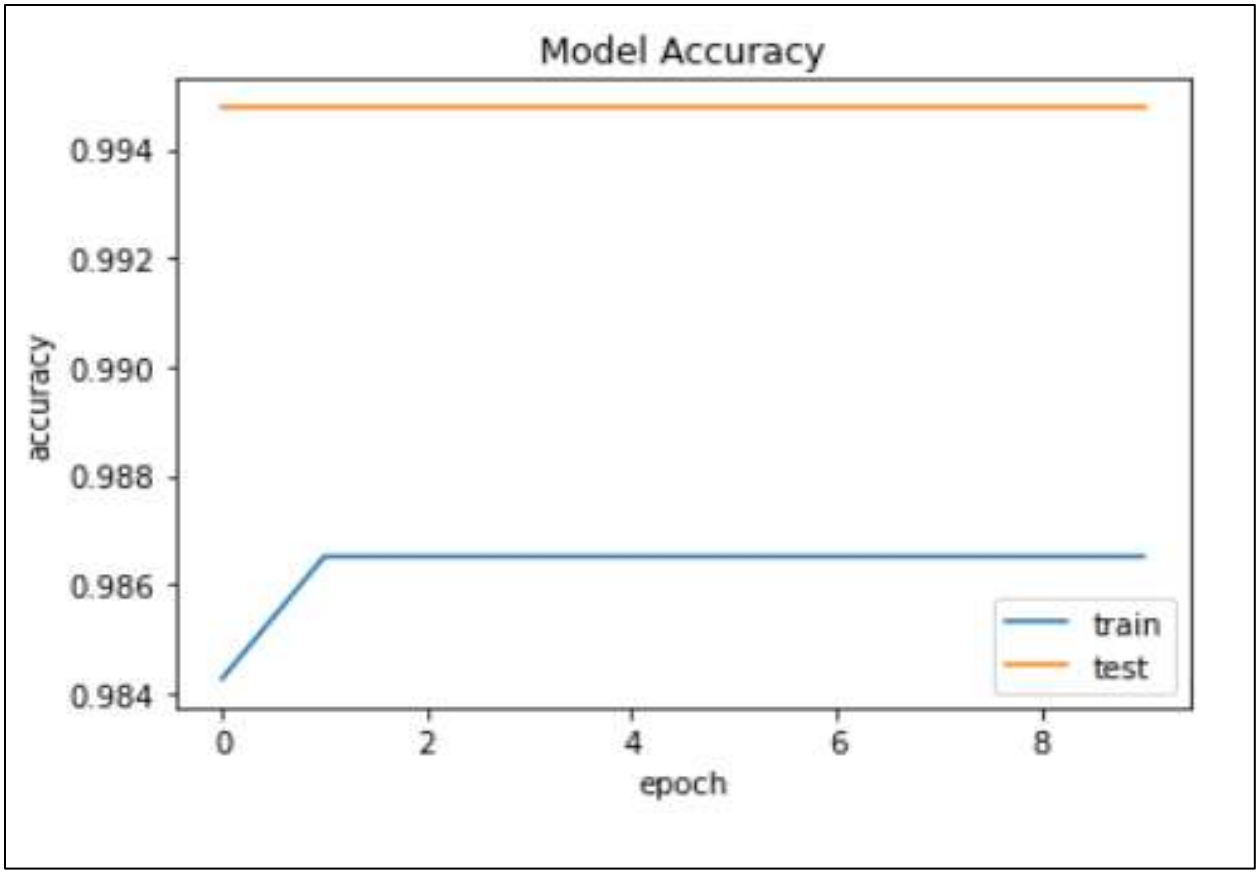
we can make full use of the data available to test the prediction performance of the ANN method as many times as possible to achieve accurate prediction performance evaluation. The ANN model we use has four hidden layers with two hidden neurons in the first layer and one neuron in the second output layer. From our experiments, such ANN configuration is found to be able to produce better prediction results compared to other ANN configurations with different numbers of hidden neurons.

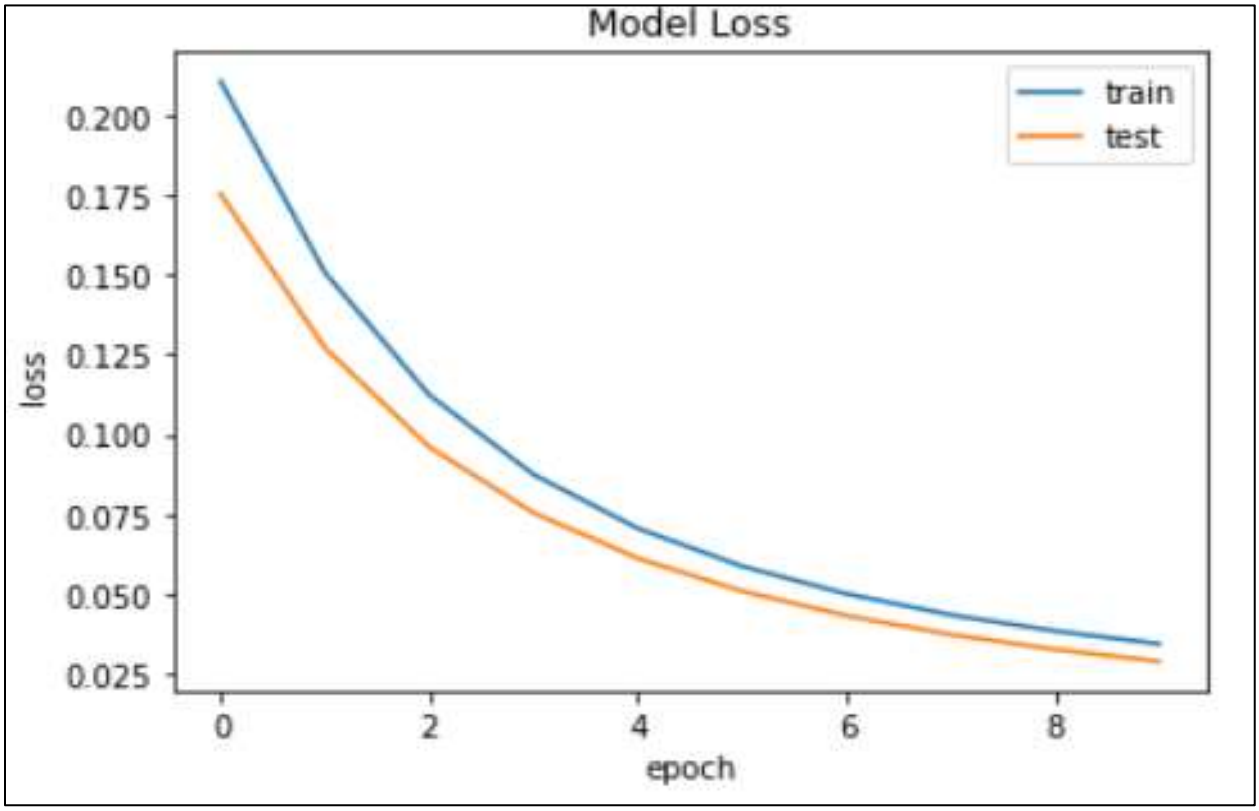
The prediction performance of the trained ANN is evaluated using the test set constructed based on the remaining data. We test the prediction performance starting every epoch, since measurement values at every point are needed to fit the measurement series and generate the fitted measurement values used as the inputs to the trained ANN model. The predicted remaining life values are obtained, and are compared with the actual life values, which are calculated based on the attribute values in the given dataset.

Plotting The Mean Absolute Error , Model Loss And Model Accuracy Graphs

The mean absolute error , model loss and model accuracy graphs are used for testing performance of our ANN Model .

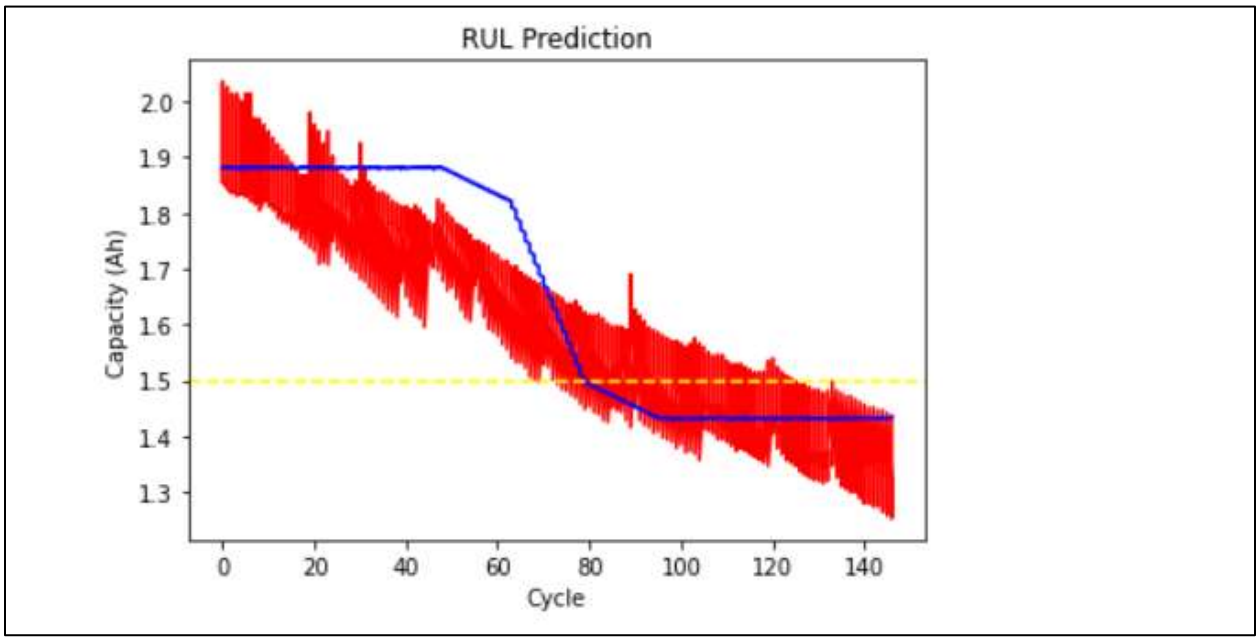



Model Accuracy
0.994
0.992
0.990
0.988
0.986
0.984
accuracy
0
2
4
6
8
epoch
train
test


Model Loss
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
loss
0
2
4
6
8
epoch
train
test

Plotting Graph Of RUL Prediction:

The Red line indicates capacity values for the battery dataset. The Blue line indicates the predicted values using NN (optimizer=sgd). The Neural network was trained on Batteries B0006, B0007, B0018 from the dataset provided by NASA However it is observed from the trails performed during implementation that prediction based on only cycle number does give accurate results.



As per the proposed methodology the RUL(remaining number of charging and discharging cycles) of the Li-ion batteries are predicted using two parameters - number of cycles of battery and its known capacity. The extrapolation techniques for neural networks have been used by giving input as the number of cycles of the given battery dataset. Using this method capacities for a particular number of cycles beyond this cycle are calculated by incrementing the cycle number.

When the capacity of particular cycle reaches the threshold capacity value for the given battery then the RUL for that battery for the given cycle number is predicted as the

difference between the current cycle number for which threshold capacity is reached and the input cycle number i.e.

Predicted RUL = Threshold cycle number - Current input cycle number For example, if the battery is currently at cycle number 10, then we will start predicting the capacities for further cycles until the threshold capacity is reached. Let's say that the threshold capacity is reached at the 20th cycle of the battery, then at present the battery has 20 - 10 = 10 cycles remaining useful life. The real RUL is calculated as the difference of the total number of life cycles a battery has and the current cycle number at which the RUL has to be predicted. Real RUL = Total no. of cycles - Current input cycle number For example, if the total number of cycles for a battery sample is 600 and we want to calculate RUL(i.e remaining number of charging and discharging cycles) for 400 th cycle then the real RUL is 600-400= 200. The RUL of testing battery i.e B0005 is predicted from capacity and cycle number.

The prediction results are based on NN methods and show the desirable properties, that is the prediction curves can converge to the real capacity curves and the RUL pdfs become narrow as the time of prediction advances. The NN method tracks the aging variation well when collecting the data after sudden changes (100 days). This is because by adjusting the model parameters using more capacity data, the NN model can effectively track the degradation trend and accordingly, achieve good prediction results.

However, the capacity prediction curves obtained by the EXP model are obviously different from the real ones for all prediction times. The reason is that the degradation characteristic for this battery is relatively complex with strong dynamics. Thus, it is difficult to be tracked and predicted using the simple EXP model.

However, the above graph was plotted by training the NN by giving all the input parameters i.e cycle number,voltage,current,temperature and time. The capacity prediction gives a decent accuracy.

Whereas if we consider all the parameters(i.e cycle number,voltage,current,temperature and time) as input to the neural network to calculate the capacity for a particular cycle then the accuracy of the model increases .

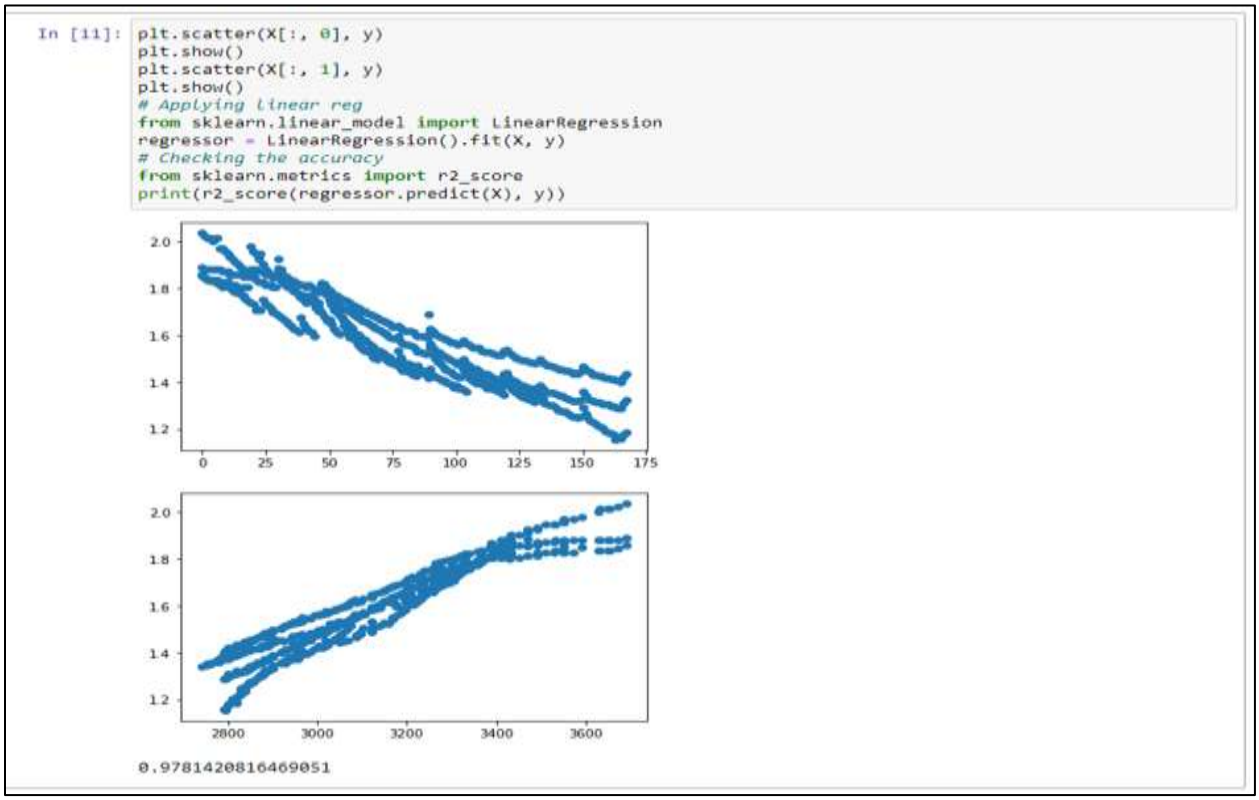
Final Conclusions Drawn:

1. Increasing no. of epochs does not necessarily improve accuracy.
2. Optimizers work in combination with loss function, a particular loss function may prove good for a given optimizer and not for others.
3. While dealing with multiple inputs (ex cycle no,voltage,current,temperature etc) , scaling the features are important.
4. For extrapolating the capacity value till the threshold capacity, multiple input neural networks fail. As an incrementing cycle no can be done, but other input values are not predictable. Hence, capacity is predicted only using the cycle no.

Thus, based on our discussion of algorithms used , we come to the conclusion that using ANN model we get accurate results .

Comparative Analysis With Other Models

To accurately estimate the remaining useful life (RUL) of the Lithium-Ion Battery remains a challenge due to the fact that lithium-ion batteries suffer charge capacity degradation. Usually, it is observed that the charge capacity degradation of the lithium-ion battery represents a linear function. Based on constant current, voltage, battery temperature, the result of our experiment to test the linearity of our data shows linear degradation of capacity, with an accuracy of ~0.98.



However, an ideal RUL prediction should be capable of taking account of non-linear behavior. The problem of overfitting arises when the prediction method solely focuses on minimizing the error or the model includes noise or fluctuated data in the learning process.

Hence, the performance of Remaining-Useful-Life prediction methods can be examined from various aspects such as:

1. **Activation Functions**
2. **Training Algorithm**
3. **Hyperparameter adjustment**
4. **Uncertainty Management**
5. **Robustness**

The selection process of an appropriate amount of data is necessary to acquire a satisfactory result. An enormous dataset may cause a huge computational burden, irrelevant of the enrichment in the training process. Keeping the hardware in check pertaining to the limited memory and computer power, it is essential to be aware of the complexity of the input and output vectors and the algorithm structure of each method. The factors on which the performance of different RUL prediction approaches are evaluated on:

- **Input Features**
- **Structure**
- **Data Calculation**

❖ **Summaries of the different criteria used for accuracy assessment**

Method	Activation Function	Training Algorithm	Hyperparameter Adjustment	Uncertainty Management	Robustness
SVM	Radial based kernel	Logistic regression and functional margin	Regularization factor SVM type regression Kernel parameter	Not Probabilistic	High robustness against small deviations
Regression	Linear	Linear Regression	_	Not Probabilistic	Robust to violations of the normality assumption
CNN	Linear	Gradient descent Backpropagation through time	Hidden Layers	Not Probabilistic	Minimum adversarial perturbation

ANN	Tanh, Sigmoid, ReLU	Gradient descent Backpropagation through time	Hidden Layers	Not Probabilistic	Robust with respect to damages of single neurons and synapses

❖ **Summaries of the different criteria for computational complexity evaluation**

Method	**Input Features and Output**	**Structure**	**Data Calculation**
SVM	Feature vector = ['Cycle', 'Time Measured(Sec)', 'Voltage Measured(V)', 'Current Measured', 'Temperature Measured', 'Capacity(Ah)'] Output = [RUL]	Radial-based kernel function	Exponential function, multiply and accumulate

Regression	Feature vector = ['Cycle', 'Time Measured(Sec)', 'Voltage Measured(V)', 'Current Measured', 'Temperature Measured', 'Capacity(Ah)'] Output = [RUL]	Explanatory variables	Exponential function, multiply and accumulate
CNN	Feature vector = ['Cycle', 'Time Measured(Sec)', 'Voltage Measured(V)', 'Current Measured', 'Temperature Measured', 'Capacity(Ah)'] Output = [RUL]	Hidden layers	Exponential functions, vector and matrix operations

ANN	Feature vector = ['Cycle', 'Time Measured(Sec)', 'Voltage Measured(V)', 'Current Measured', 'Temperature Measured', 'Capacity(Ah)'] Output = [RUL]	Hidden layers	Exponential functions, vector and matrix operations

Based on the previous summary, the advantages and disadvantages of the proposed methods are compared. SVM has satisfactory performance in nonlinear and high-dimensional models, can deal with local minima and small sample sizes, and has a short calculation time. However, it cannot express uncertainty due to its difficulty with calculating kernel and regularization parameters. Regression models cannot work properly if the input data has errors (that is poor quality data). If the data preprocessing is not performed well to remove missing values or redundant data or outliers or imbalanced data distribution, the validity of the regression model suffers. While regression is a useful tool for analysis, it does have its disadvantages, including its sensitivity to outliers and more. CNN comes ahead strong, compared to its predecessors, in that it automatically detects the important features without any human supervision. However, a Convolutional neural network is significantly slower due to an operation such as maxpool. If the CNN has several layers then the training process takes a lot of time if the computer doesn't consist of a good GPU. Artificial neural networks are algorithms that can be used to perform nonlinear statistical modeling and provide a new alternative to logistic regression, the most suitable in our case for developing predictive models for RUL prediction. Neural networks offer a number of advantages, including requiring less formal statistical training, ability to

implicitly detect complex nonlinear relationships between dependent and independent variables, ability to detect all possible interactions between predictor variables, and the availability of multiple training algorithms.

Method	**Advantages**	**Disadvantages**
SVM	● Satisfactory performance in non-linear and high-dimension models. ● Capable of dealing with the local minimum, non-linear, and small sample size problems ● The global optimal solution can be obtained ● High prediction accuracy ● Less prediction times ● Non-parametric	● Kernel and regularization parameters are difficult to calculate ● Low expression of uncertainty. ● Kernel functions need to satisfy the Mercer criterion

Regression	• Fitness measured in terms of the correlation coefficients • Satisfactory predictive power • Inclusive of all variables • Pervasive.	• Susceptible to errors in data. • Susceptible to collinear problems. • Number of variables increases and the reliability of the regression models decreases. • Do not take care of nonlinearity.
CNN	• Very High accuracy in image recognition problems. • Automatically detects the important features without any human supervision. • Weight sharing.	• Do not encode the position and orientation of objects. • Lack of ability to be spatially invariant to the input data. • Lots of training data is required.

ANN	• Problems represented by attribute-value pairs. • Output may be discrete-valued, real-valued, or a vector of several real or discrete-valued attributes. • Robust to noise in the training data. • Fast evaluation of the learned target function • Able to bear long training times	• Hardware Dependence • No specific rule for determining the structure of artificial neural networks. • Difficulty of showing the problem to the network • The duration of the network is unknown

Comparison:

❖ **Between ANN and CNN:**

Models	**ANN**	**CNN**
Type of Data	Tabular Data, Text Data	Image Data
Parameter Sharing	No	Yes
Fixed Length input	Yes	Yes
Recurrent Connections	No	No
Vanishing and Exploding Gradient	Yes	Yes
Spatial Relationship	No	Yes
Performance	Less powerful	More powerful
Application	Facial recognition and Computer vision	Facial recognition, text digitization and Natural language processing
Main advantages	Having fault tolerance, Ability to work with incomplete knowledge	High accuracy in image recognition problems, Weight sharing
Disadvantages	Hardware dependence, Unexplained behavior of the network	Large training data needed, don’t encode the position and orientation of object

❖ **Between ANN and SVM:**

Models	**ANN**	**SVM**
Accuracy	Average	Good
Model complexity	Medium	High
Computation speed	Medium	Medium
Computation difficulty	Medium	Medium
Energy sampling type	Long-term; short-term	Long-term; short-term
Determination of parameters/models	Average	Simple

❖ **Between ANN and Regression:**

Models	**ANN**	**Regression**
Model building	Requires less statistical knowledge	Requires more statistical knowledge
Ability to detect complex relationships	Automatically models any complex relationships between input and output variables	Fails to detect complex relationships between input and output variables unless stated by the modeler
Ability to detect interactions	Can detect implicit interactions among predictor variables	Requires explicit modeling of interactions

Generalizability and overfitting	Prone to overfitting	Overfitting is less of a concern
Discrimination ability	Good in general	Good in general
Computational time	More required	Less required
Sharing the models with other researchers	Existing model is not easily shared	Existing model is easily shared
Confidence intervals	Difficult to calculate	Easy to calculate
Clinical interpretations for the decision maker	Black box i.e. difficult to identify important predictors	Easy to identify important predictors

Comparison based on the code:

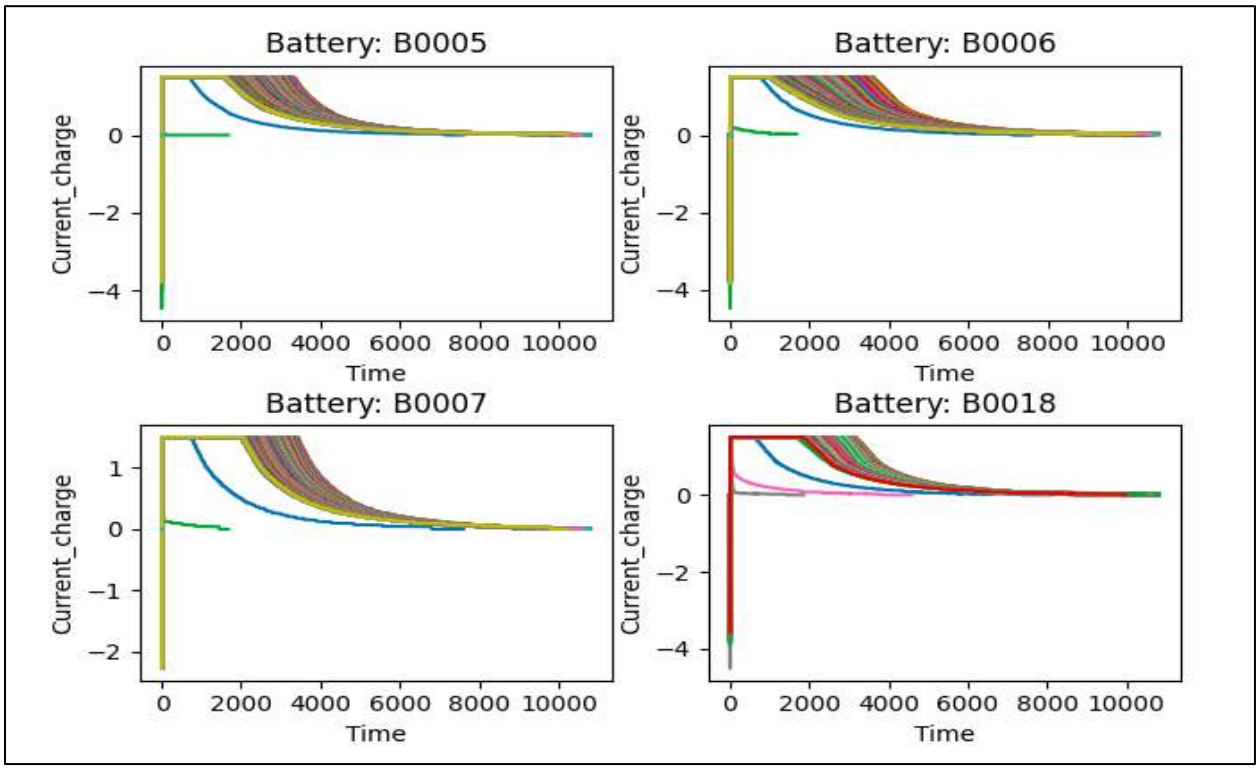
- **REGRESSION**

Data Analysis

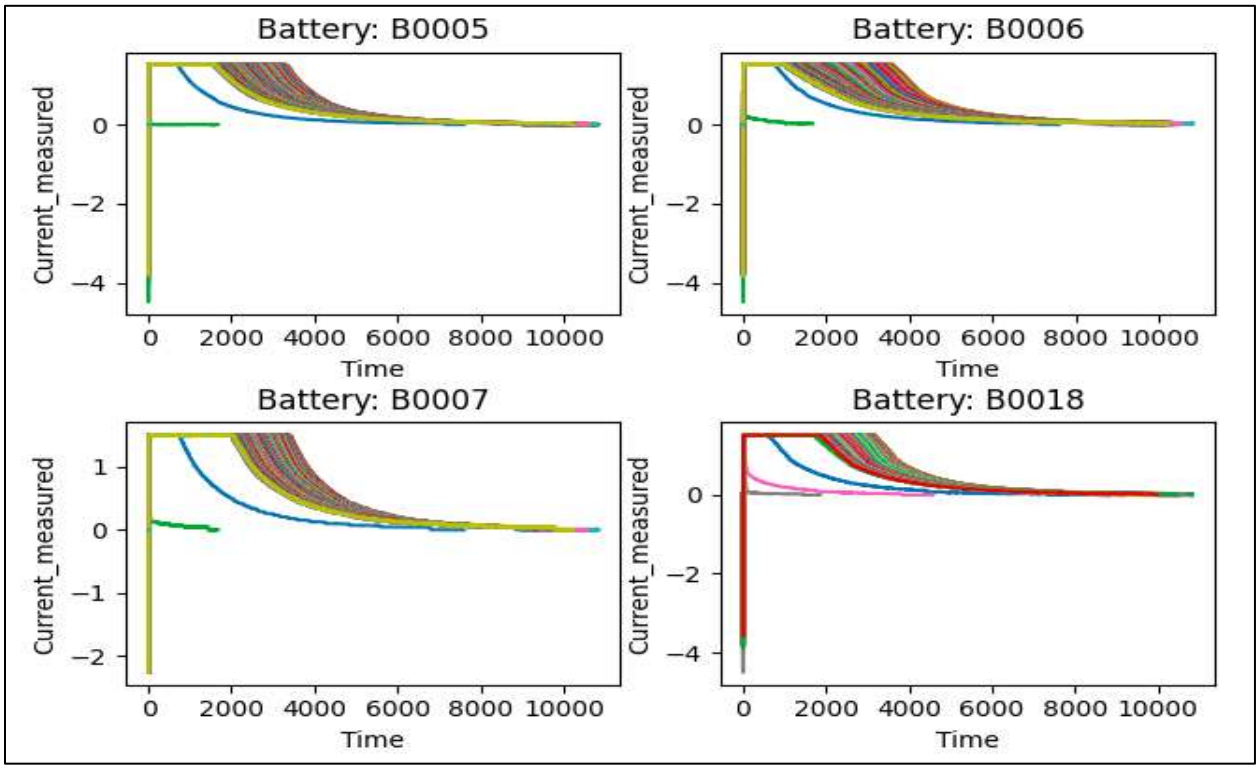
Charge all cycles:

```
params = ['Voltage_measured', 'Current_measured', 'Temperature_measured', 'Current_charge', 'Voltage_charge']

for p in params:
    fig, axs = plt.subplots((len(bs) + 1) // 2, 2)
    param = p
    for i in range(len(bs)):
        for j in range(datas[i].size):
            if types[i][j] == 'charge':

                if i % 2 == 0:
                    axs[i // 2, 0].plot(datas[i][j]['Time'][0][0][0], datas[i][j][param][0][0][0])
                    axs[i // 2, 0].set_title(f'Battery: {bs[i]}')
                else:
                    axs[i // 2, 1].plot(datas[i][j]['Time'][0][0][0], datas[i][j][param][0][0][0])
                    axs[i // 2, 1].set_title(f'Battery: {bs[i]}')
    for ax in axs.flat:
        ax.set(ylabel = param, xlabel = 'Time')
    fig.tight_layout(pad = 0.3)
    fig.savefig(f'PLOTS/Charge/All/{p}_all.png')
```

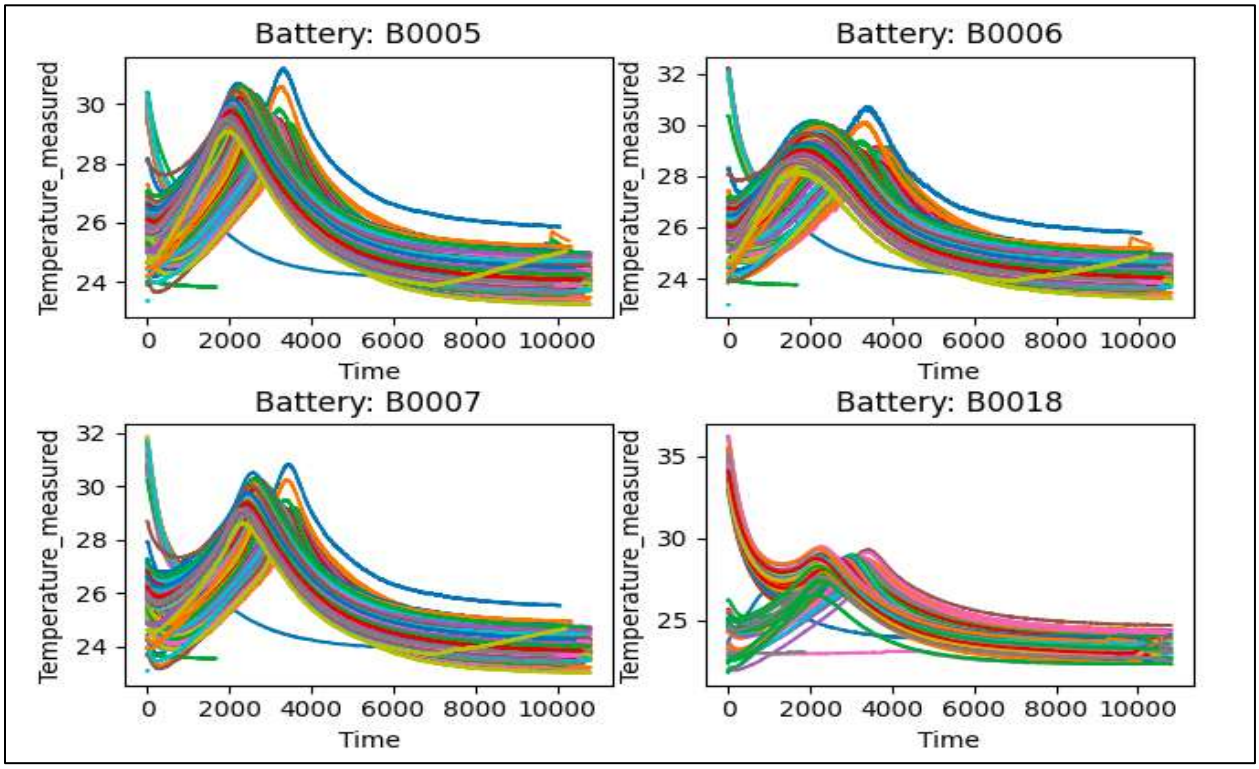
Current charge all:



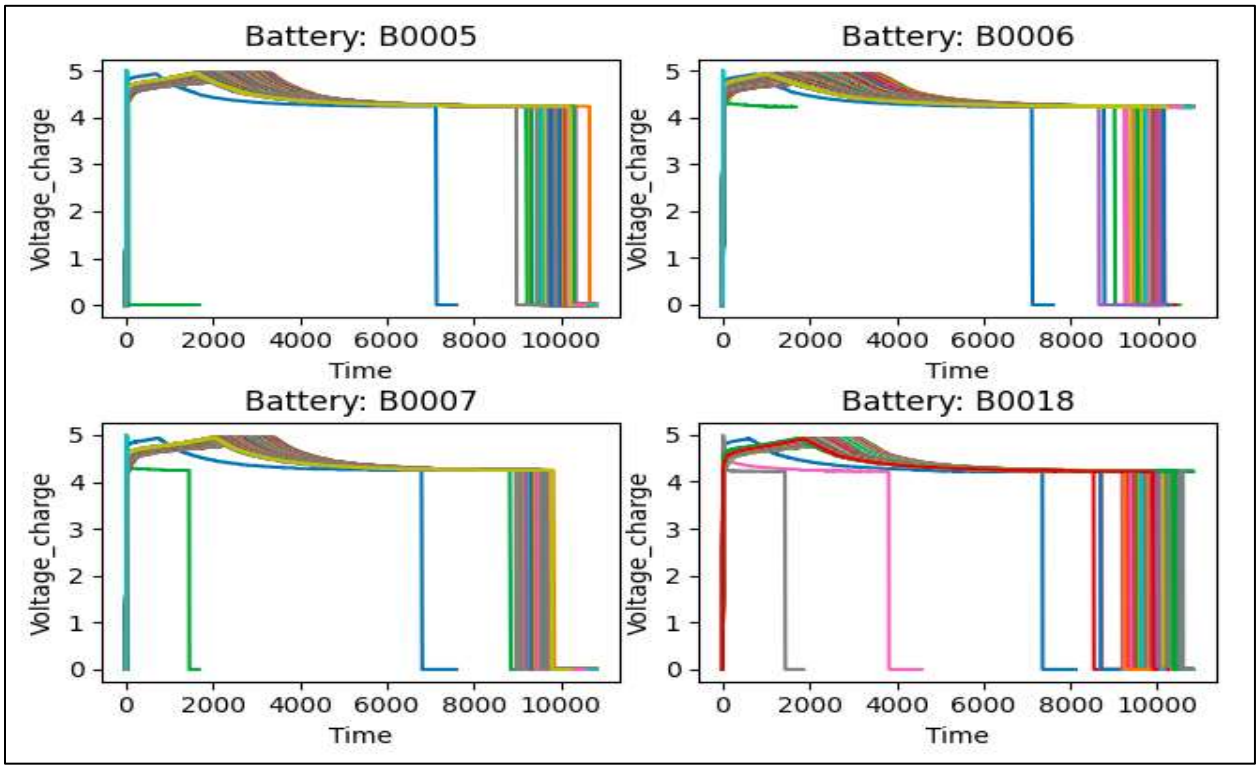
Current measured all:



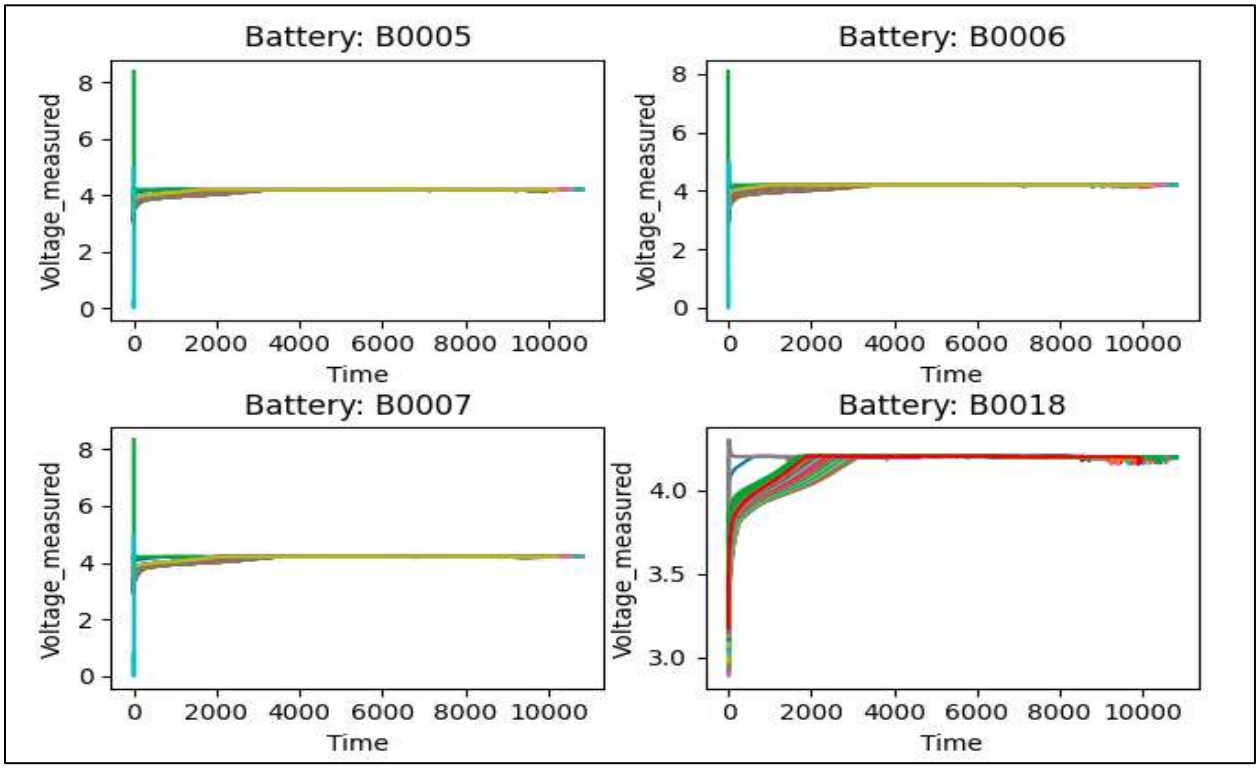
Temperature measured all:



Voltage charge all:



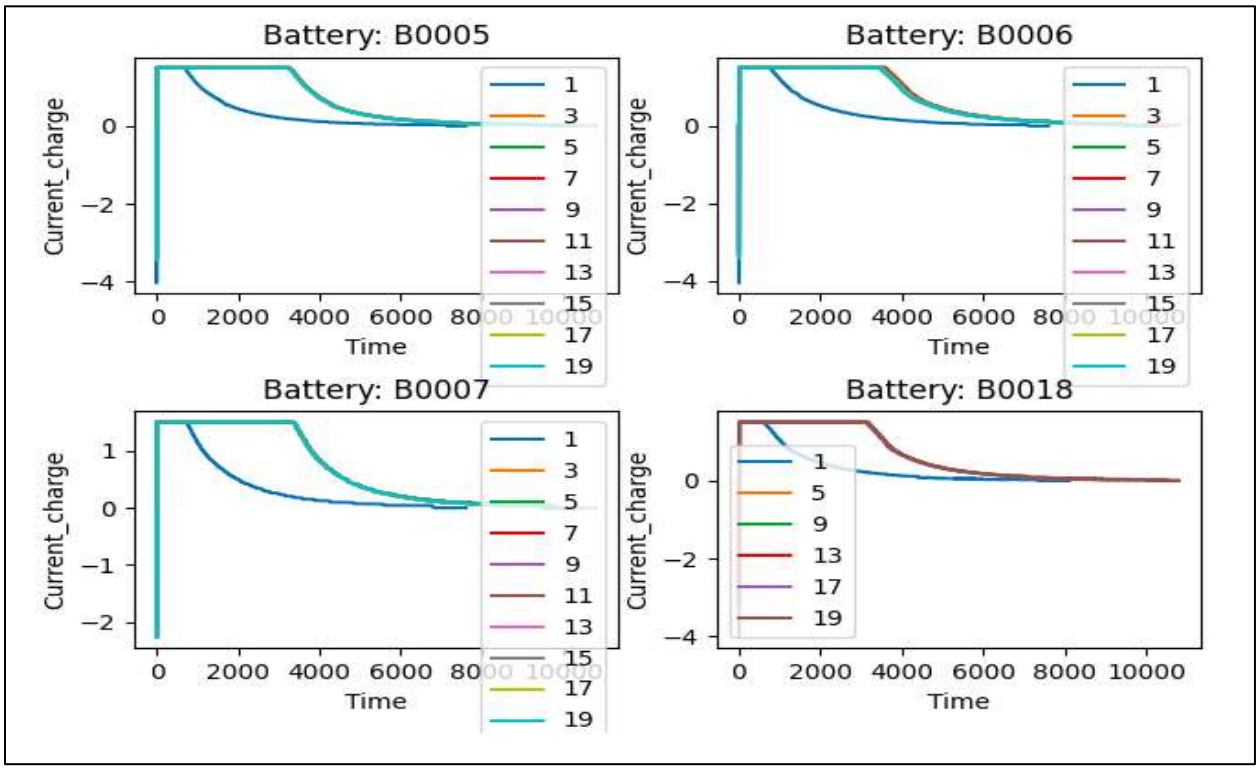
Voltage measured all:



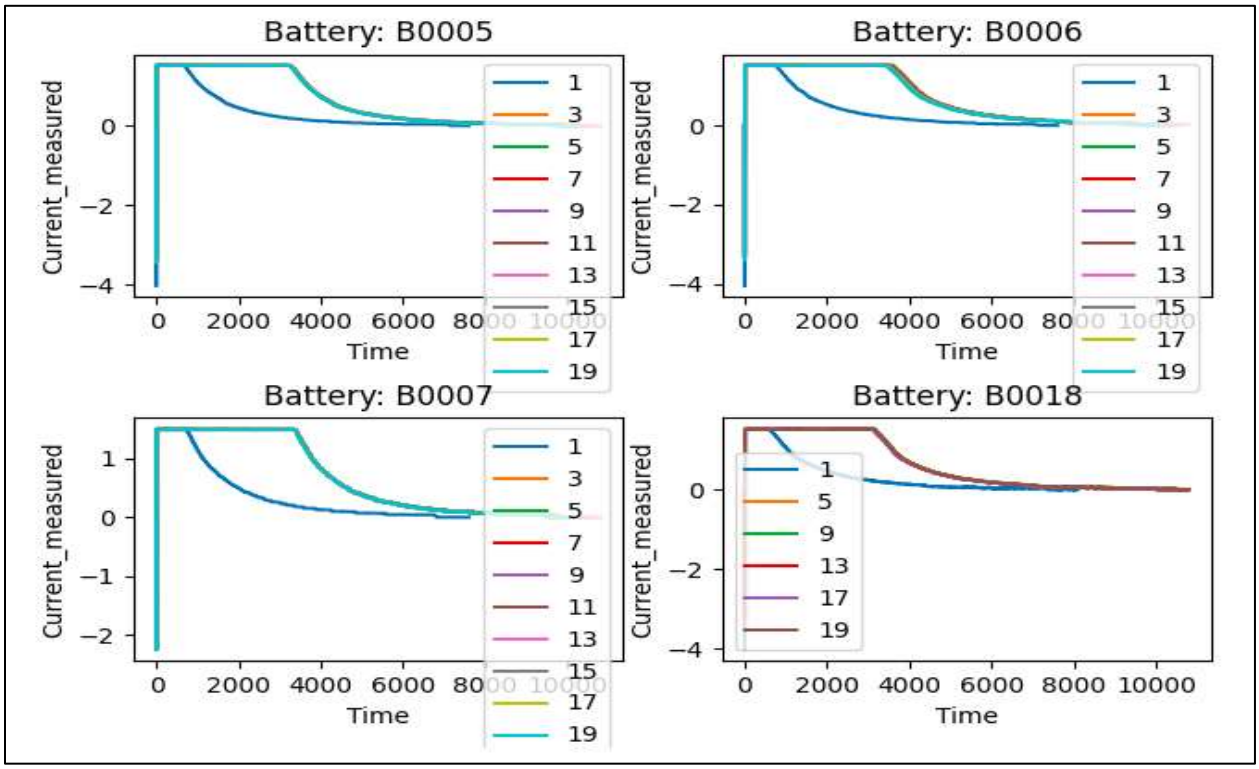
Charge first and last cycles:

```
for p in params:

    # Printing first cycles

    fig, axs = plt.subplots((len(bs) + 1) // 2, 2)
    param = p
    for i in range(len(bs)):
        for j in range(20):
            if types[i][j] == 'charge':
                if i % 2 == 0:
                    axs[i // 2, 0].plot(datas[i][j]['Time'][0][0][0], datas[i][j][param][0][0][0], label = f'{j + 1}')
                    axs[i // 2, 0].set_title(f'Battery: {bs[i]}')
                    axs[i // 2, 0].legend()
                else:
                    axs[i // 2, 1].plot(datas[i][j]['Time'][0][0][0], datas[i][j][param][0][0][0], label = f'{j + 1}')
                    axs[i // 2, 1].set_title(f'Battery: {bs[i]}')
                    axs[i // 2, 1].legend()
    for ax in axs.flat:
        ax.set(ylabel = param, xlabel = 'Time')
    fig.tight_layout(pad = 0.3)
    fig.savefig(f'PLOTS/Charge/First/{p}_first.png')
    # Printing last cycles

    fig, axs = plt.subplots((len(bs) + 1) // 2, 2)
    for i in range(len(bs)):
        for j in range(datas[i].size - 20, datas[i].size):
            if types[i][j] == 'charge':
                if i % 2 == 0:
                    axs[i // 2, 0].plot(datas[i][j]['Time'][0][0][0], datas[i][j][param][0][0][0], label = f'{j + 1}')
                    axs[i // 2, 0].set_title(f'Battery: {bs[i]}')
                    axs[i // 2, 0].legend()
                else:
                    axs[i // 2, 1].plot(datas[i][j]['Time'][0][0][0], datas[i][j][param][0][0][0], label = f'{j + 1}')
                    axs[i // 2, 1].set_title(f'Battery: {bs[i]}')
                    axs[i // 2, 1].legend()

    for ax in axs.flat:
        ax.set(ylabel = param, xlabel = 'Time')
    fig.tight_layout(pad = 0.3)
    fig.savefig(f'PLOTS/Charge/Last/{p}_last.png')
```

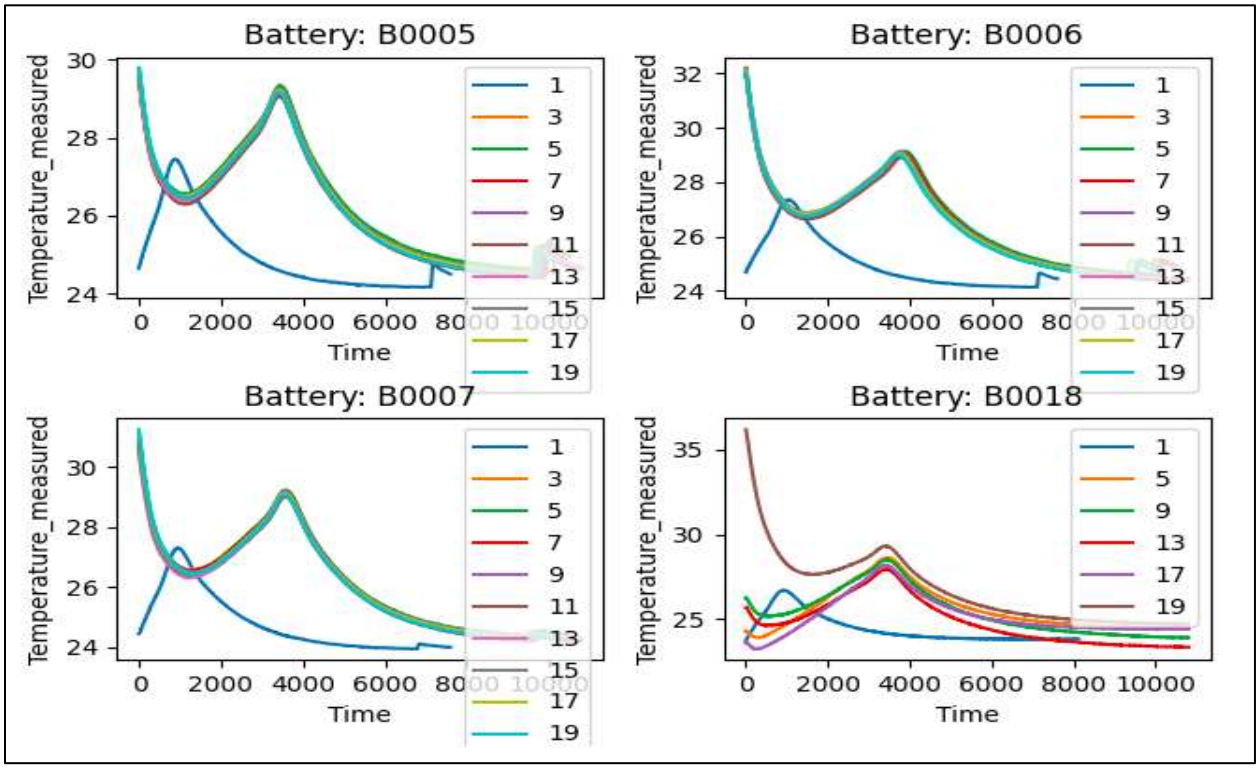
Current charge first:



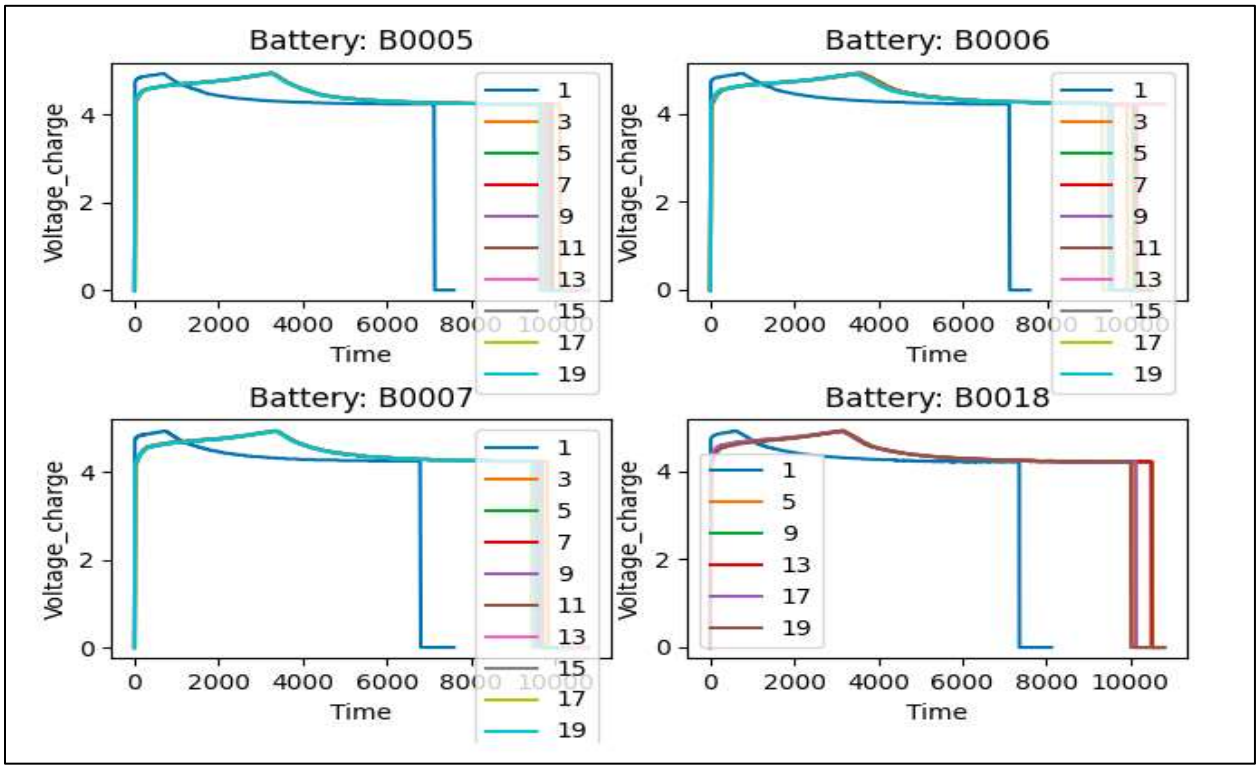
Current measured first:



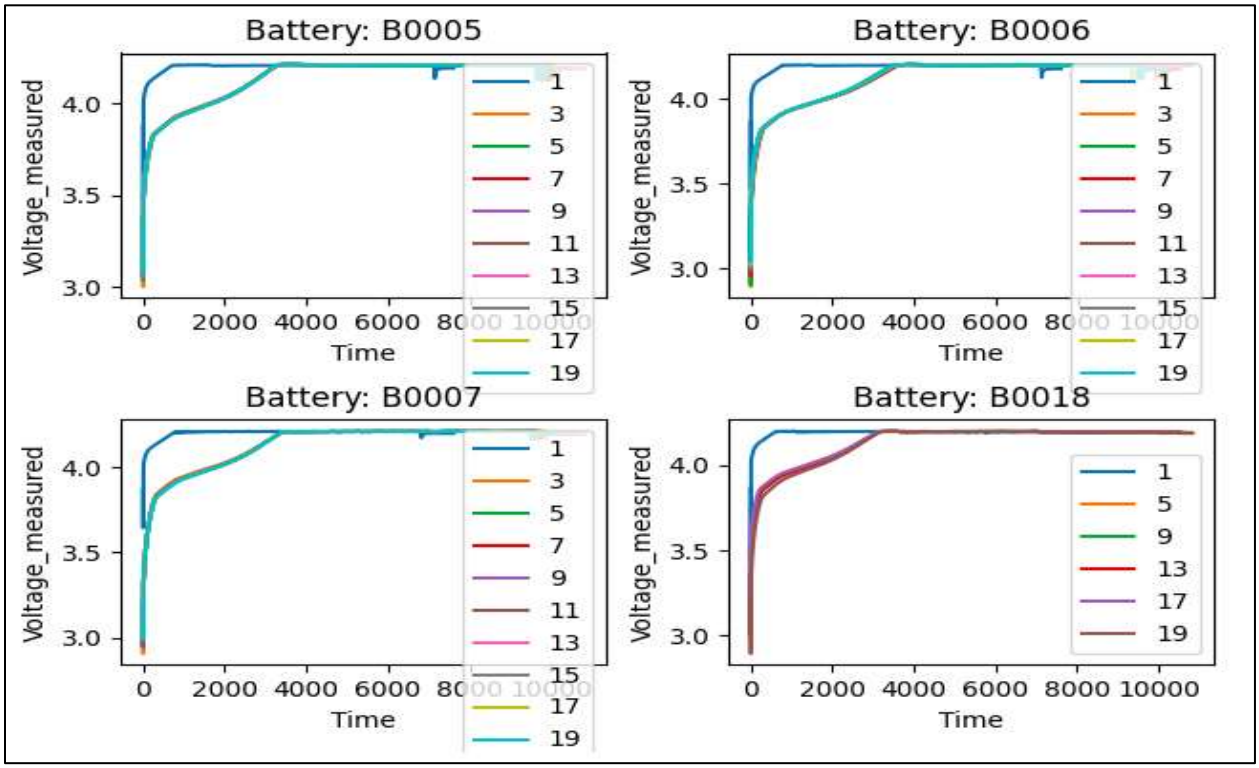
Temperature measured first:



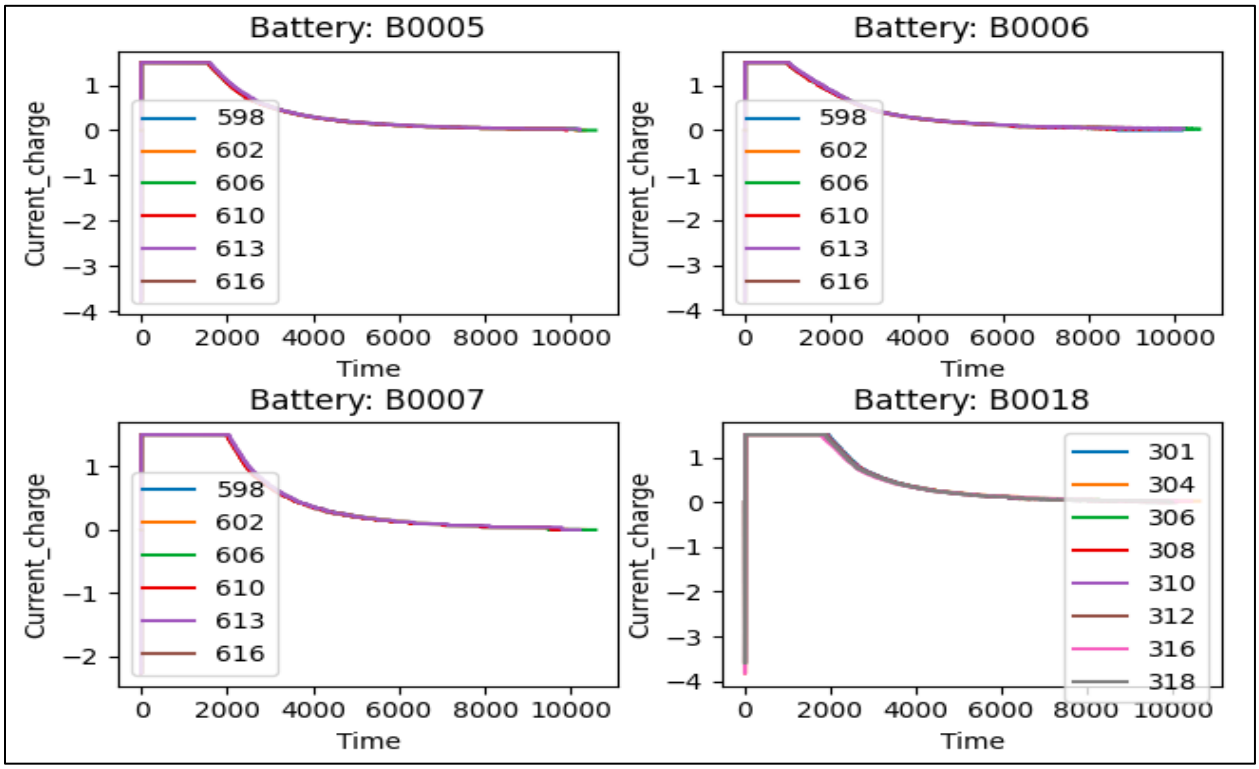
Voltage charge first:



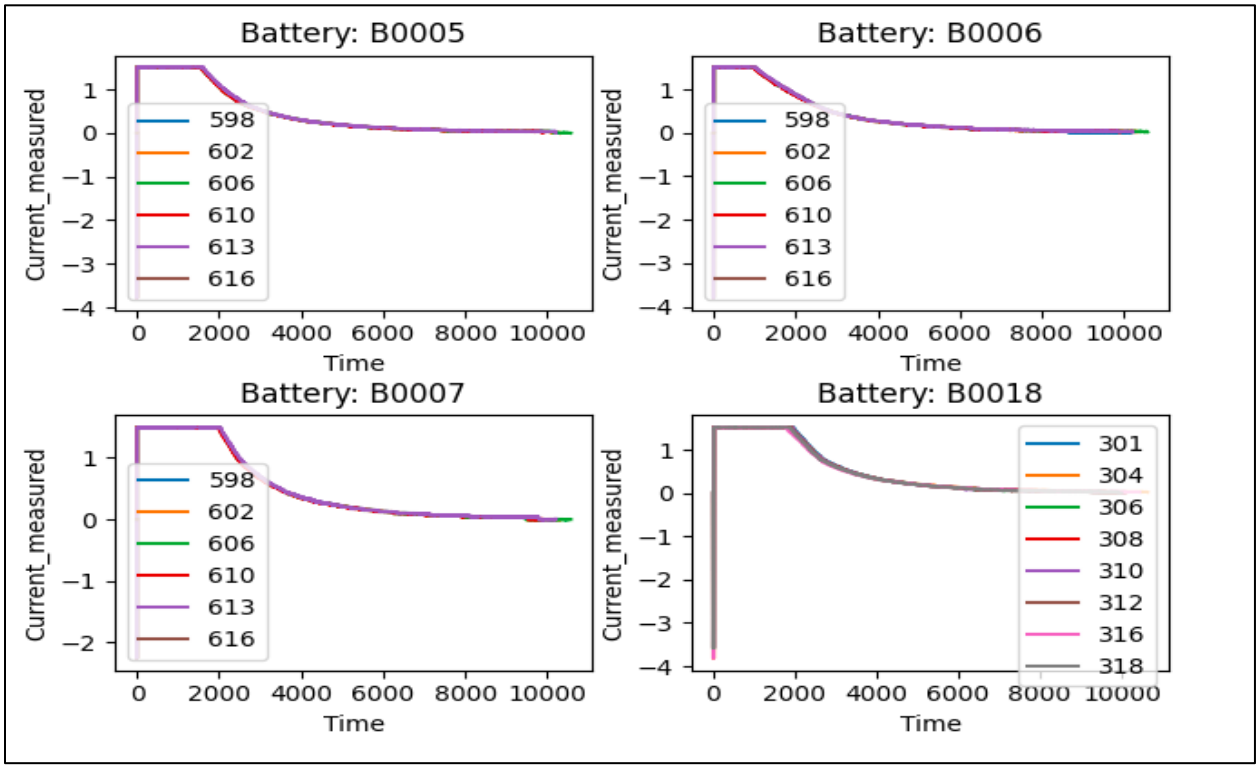
Voltage measured first:



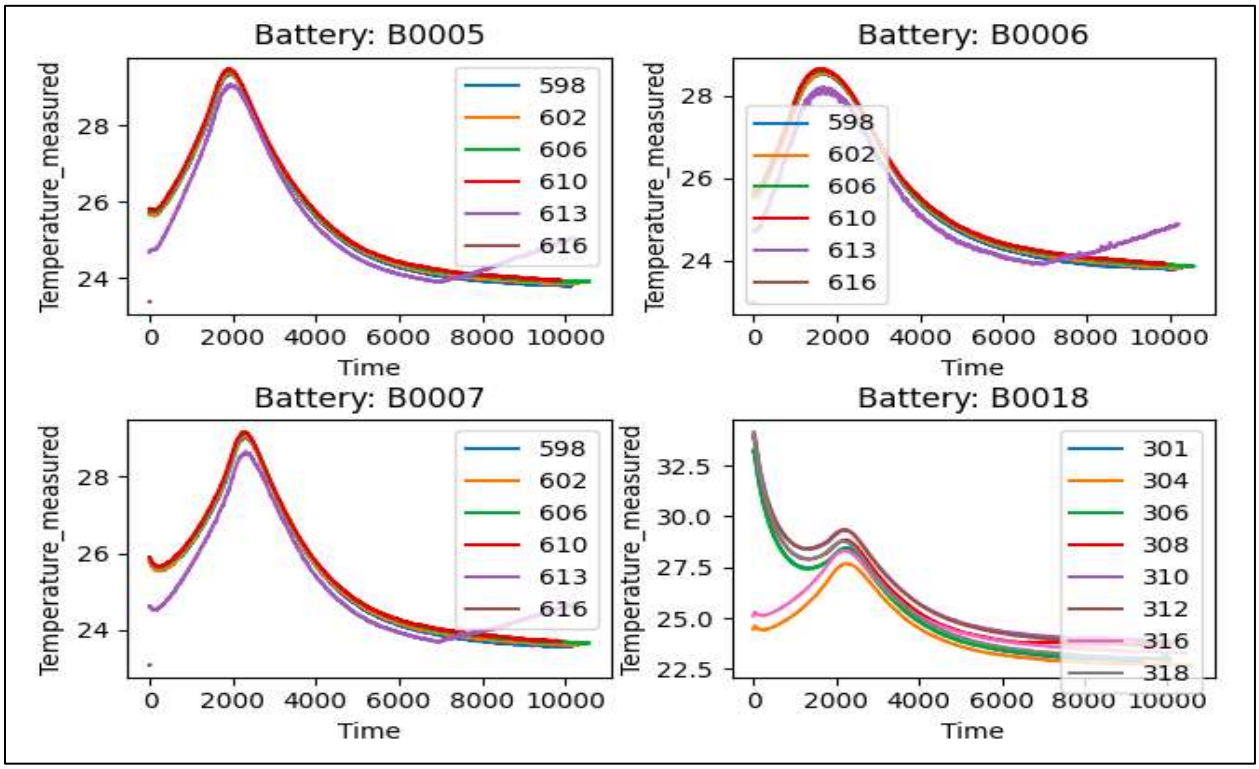
Current charge last:



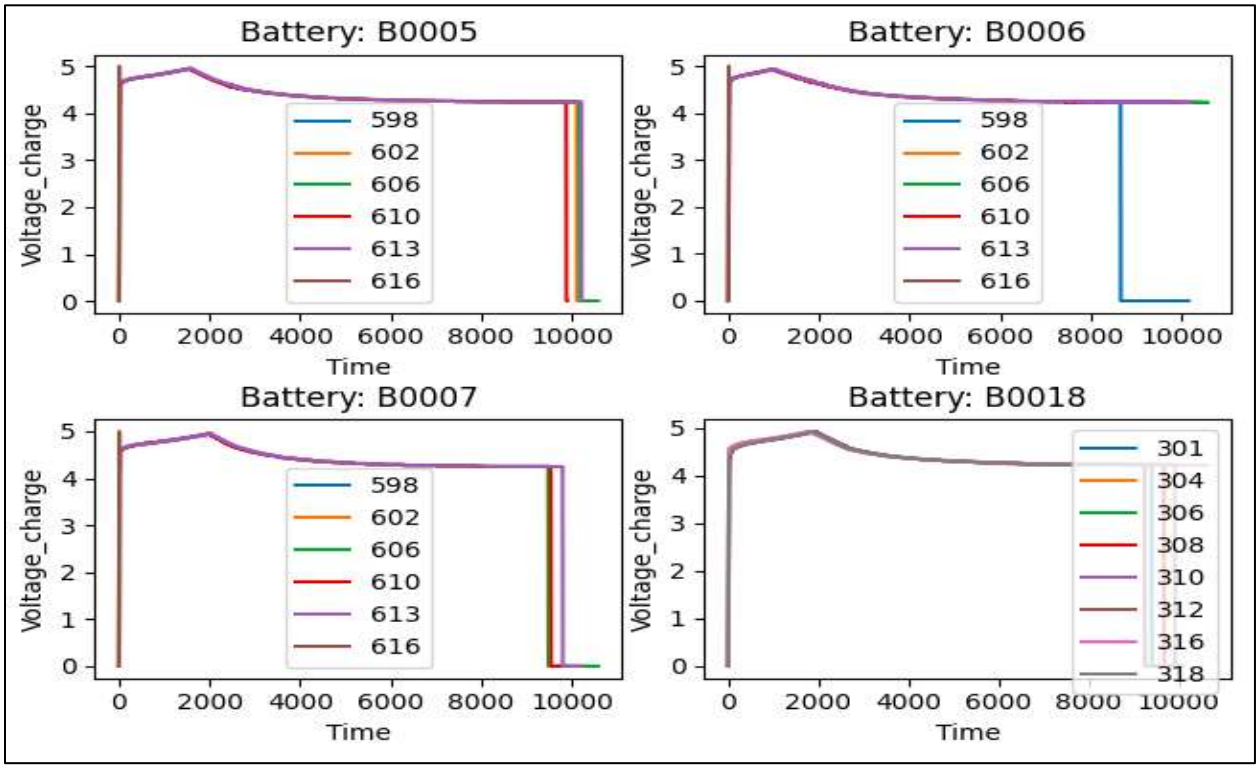
Current measured last:



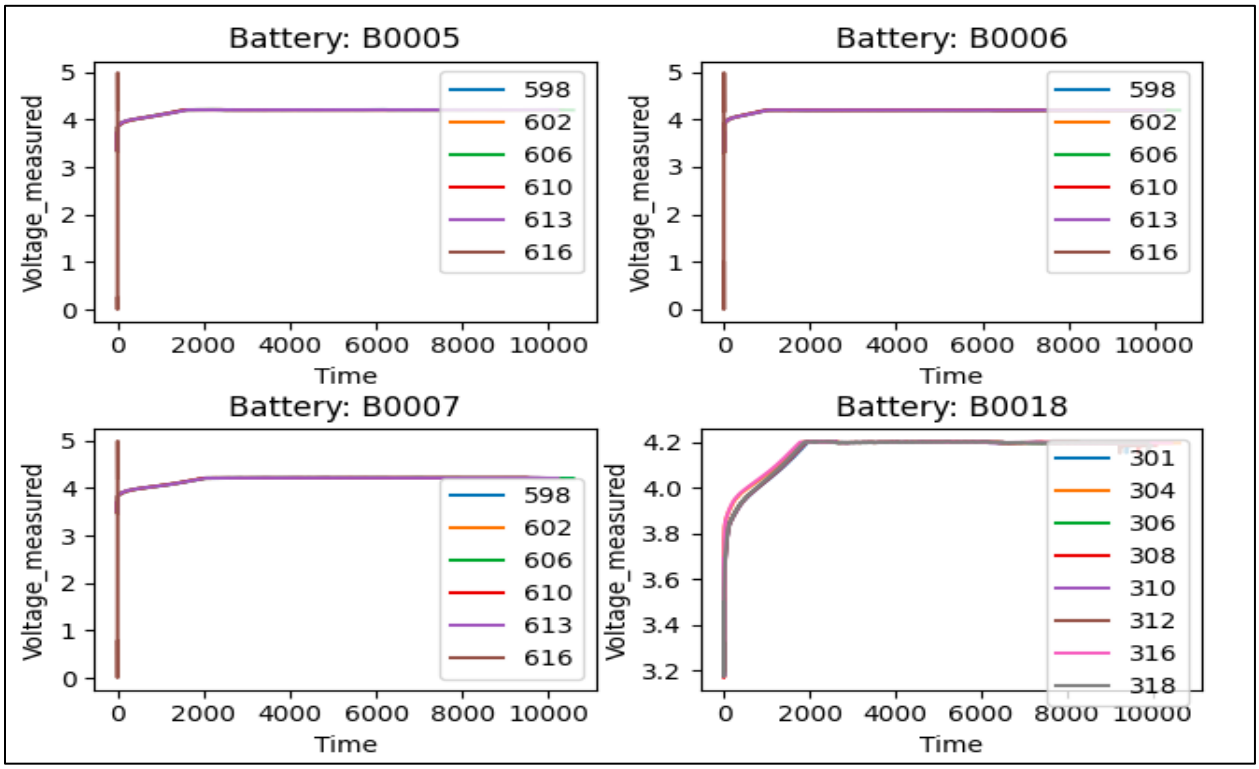
Temperature measured last:



Voltage charge last:



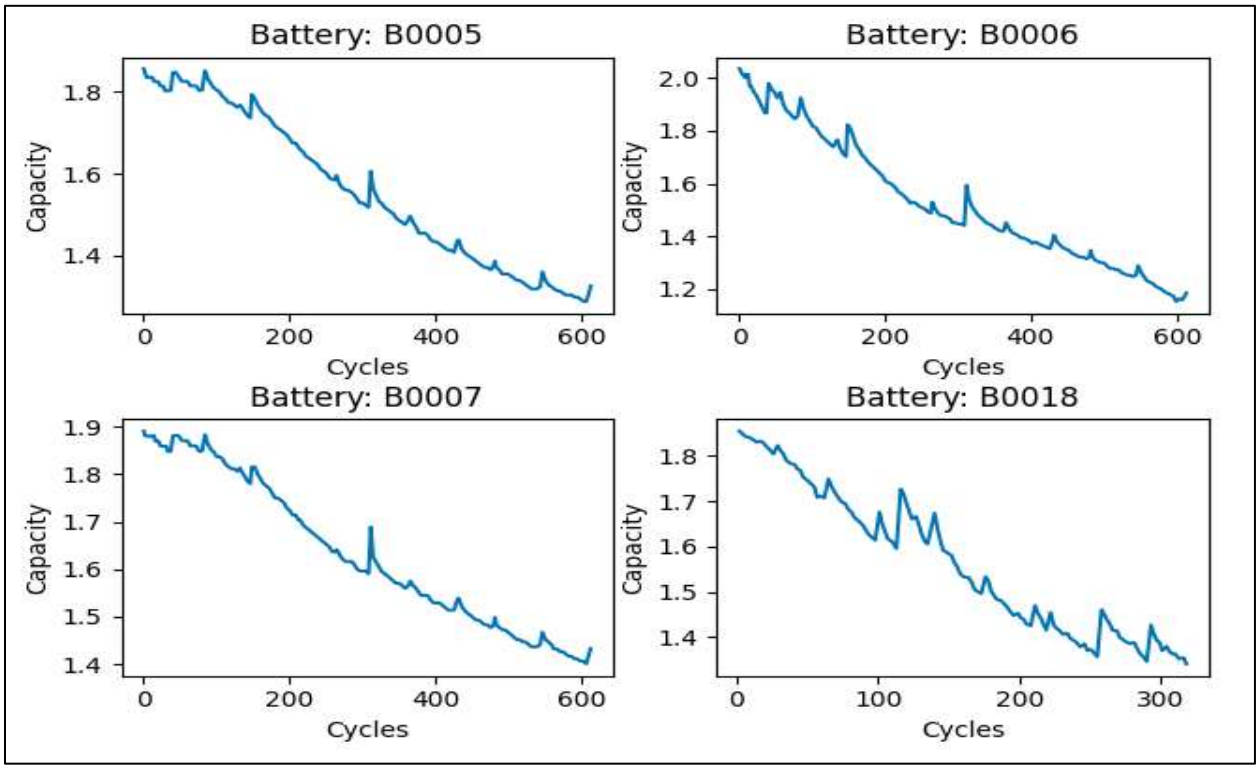
Voltage measured last:



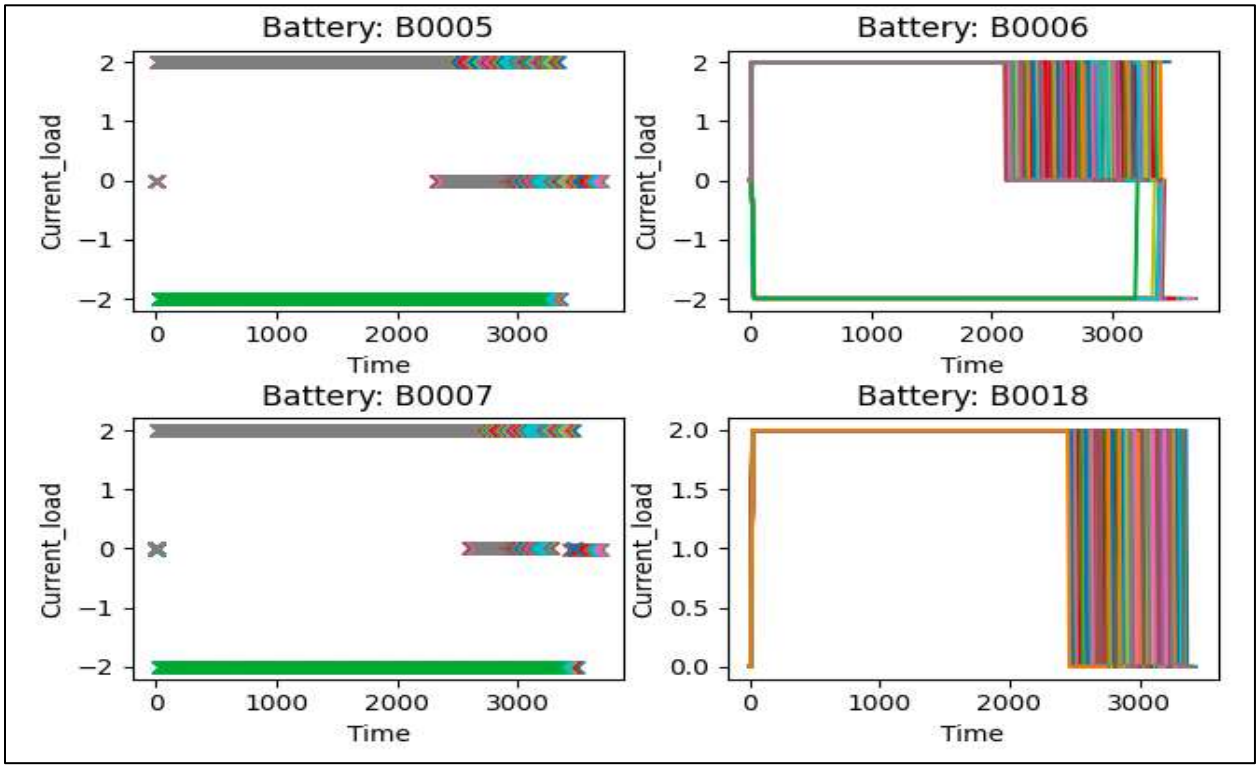
Discharge all cycles:

```
params = ['Voltage_measured', 'Current_measured', 'Temperature_measured', 'Current_load', 'Voltage_load']

for param in params:
    fig, axs = plt.subplots((len(bs) + 1) // 2, 2)
    for i in range(len(bs)):
        for j in range(datas[i].size):
            if types[i][j] == 'discharge':
                if i % 2 == 0:
                    axs[i // 2, 0].plot(datas[i][j]['Time'][0][0][0], datas[i][j][param][0][0][0], 'x')
                    axs[i // 2, 0].set_title(f'Battery: {bs[i]}')
                else:
                    axs[i // 2, 1].plot(datas[i][j]['Time'][0][0][0], datas[i][j][param][0][0][0])
                    axs[i // 2, 1].set_title(f'Battery: {bs[i]}')
    for ax in axs.flat:
        ax.set(ylabel = param, xlabel = 'Time')
    fig.tight_layout(pad = 0.3)
    fig.savefig(f'PLOTS/Discharge/All/{param}_all.png')

fig, axs = plt.subplots((len(bs) + 1) // 2, 2)
for i in range(len(bs)):
    cap = []
    cycle = []
    for j in range(datas[i].size):
        if types[i][j] == 'discharge':
            cap.append(datas[i][j]['Capacity'][0][0][0][0])
            cycle.append(j)
    if i % 2 == 0:
        axs[i // 2, 0].plot(cycle, cap)
        axs[i // 2, 0].set_title(f'Battery: {bs[i]}')
    else:
        axs[i // 2, 1].plot(cycle, cap)
        axs[i // 2, 1].set_title(f'Battery: {bs[i]}')

    for ax in axs.flat:
        ax.set(ylabel = 'Capacity', xlabel = 'Cycles')
fig.tight_layout(pad = 0.3)
fig.savefig(f'PLOTS/Discharge/Capacity_Line.png')
```

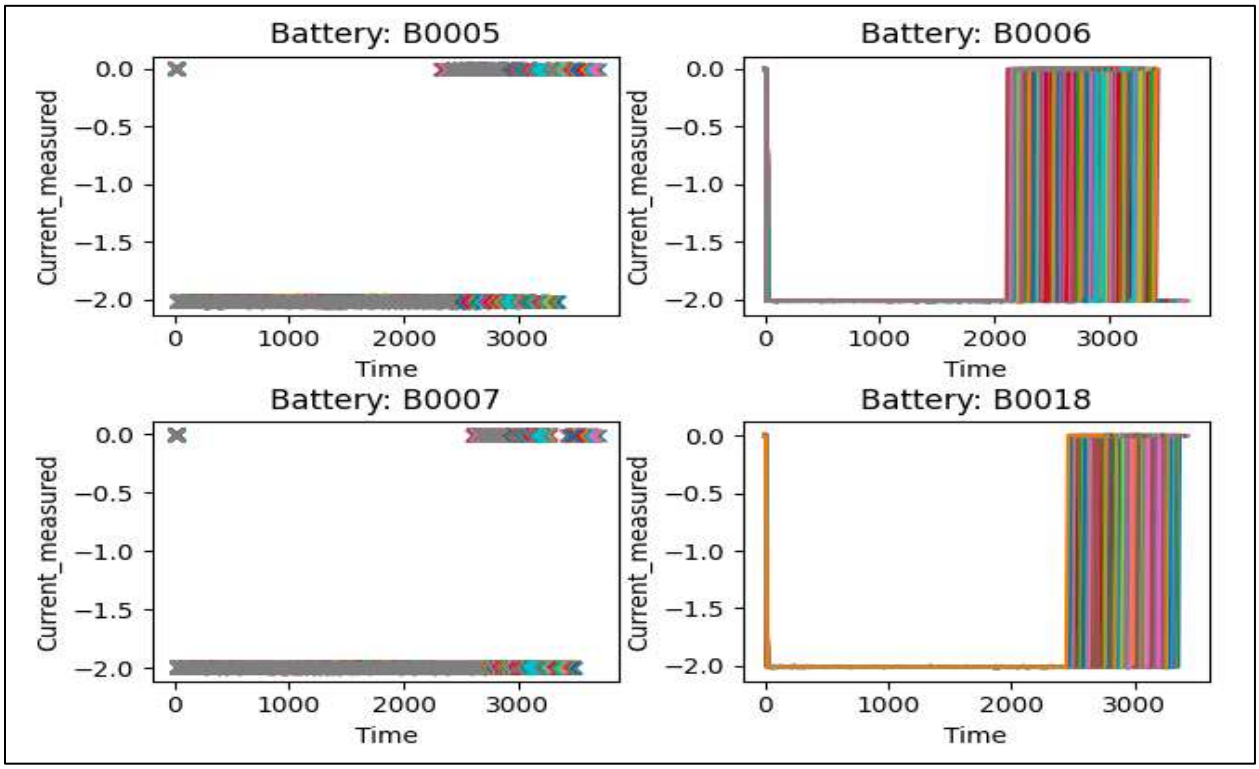
Capacity line:



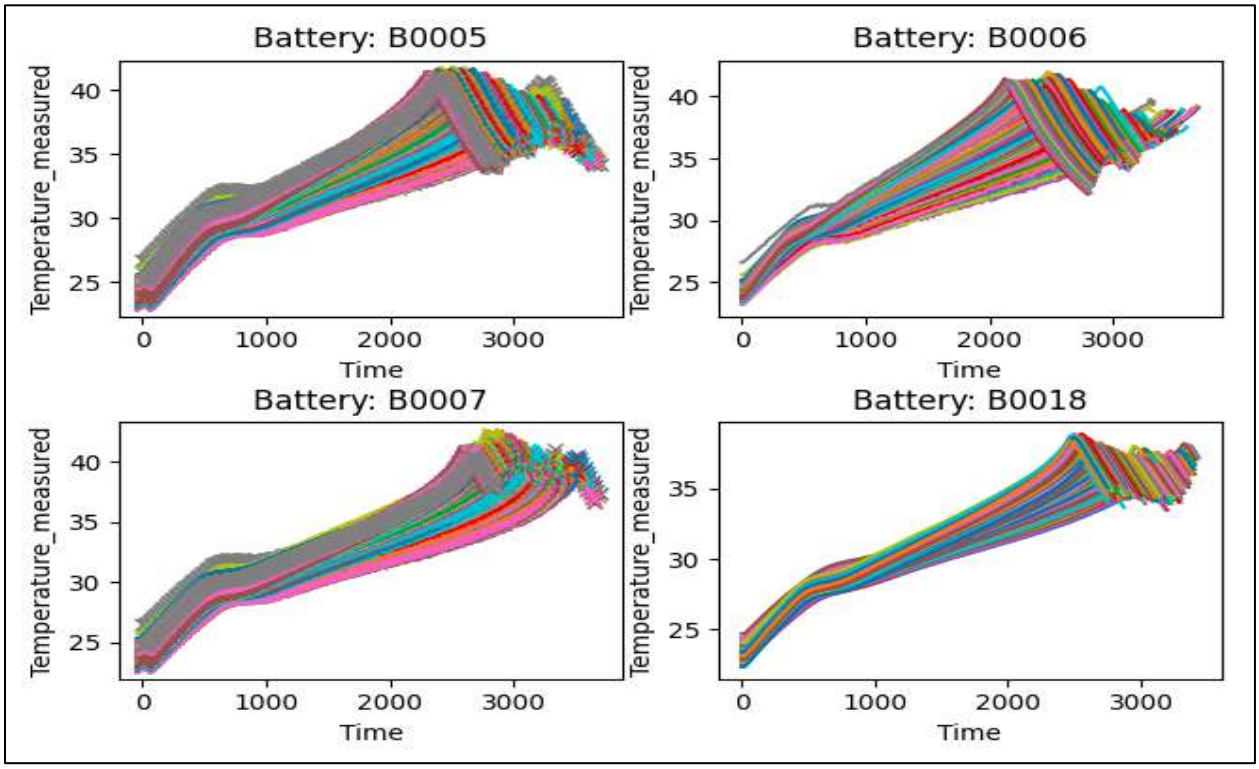
Current load all:



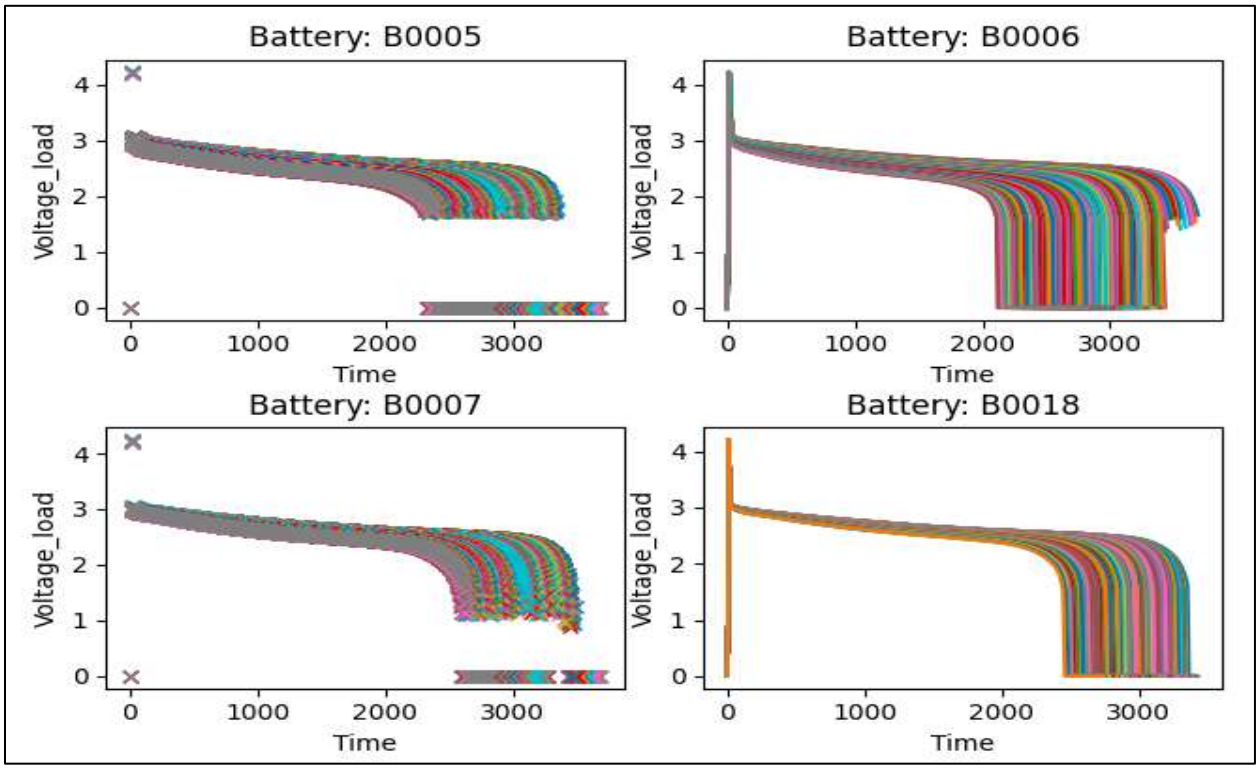
Current measured all:



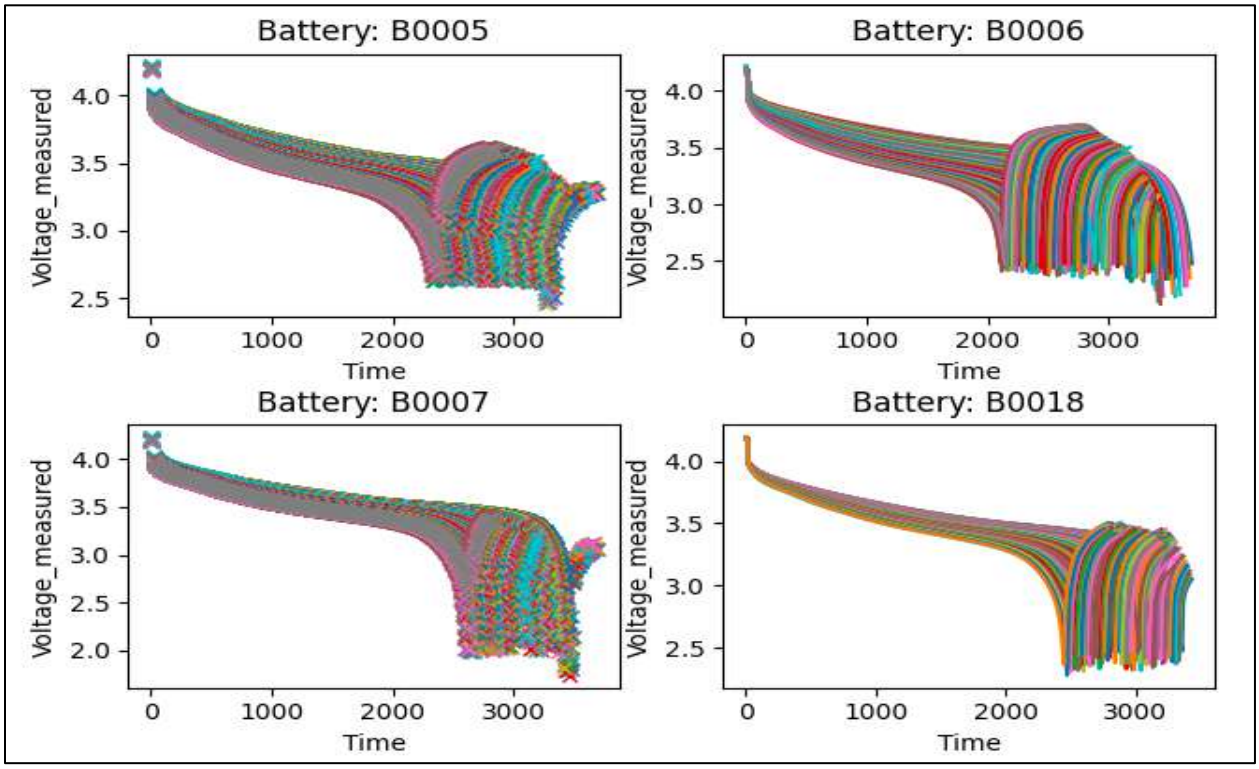
Temperature measured all:



Voltage load all:



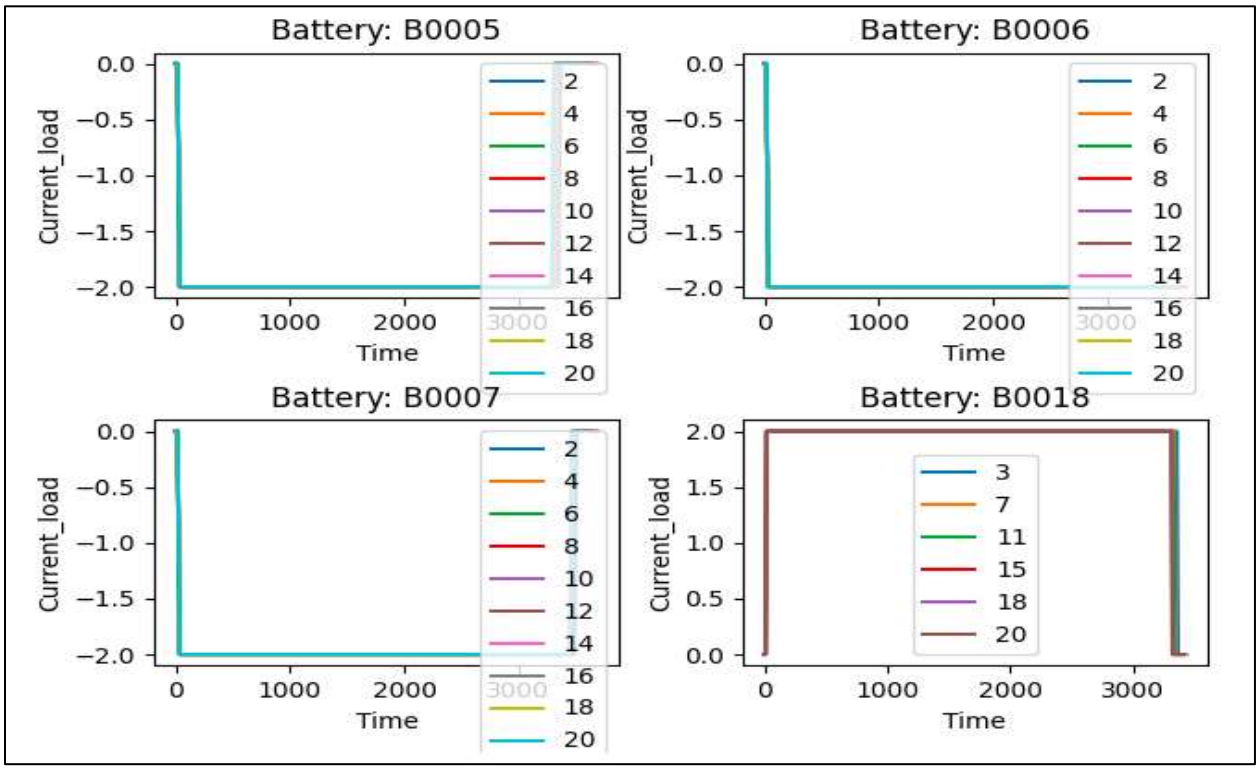
Voltage measured all:



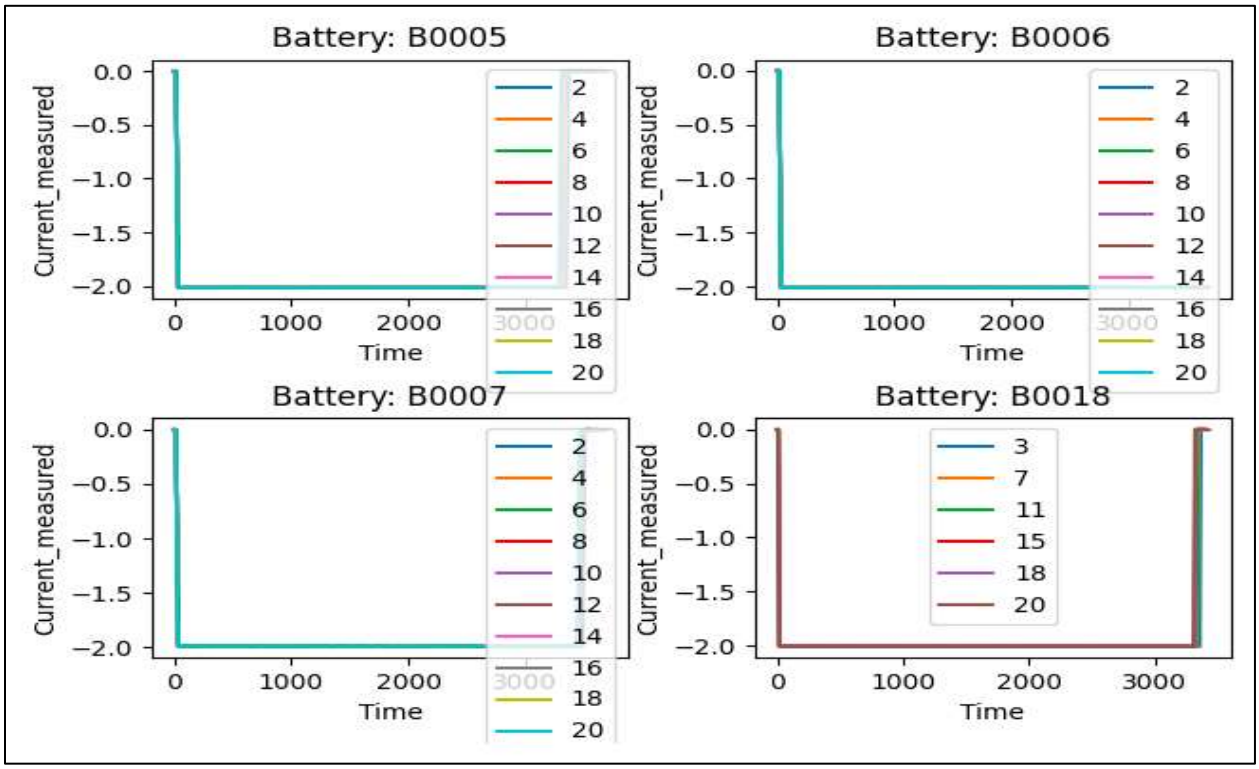
Discharge first and last cycles:

```
for p in params:

    # Printing first cycles

    fig, axs = plt.subplots((len(bs) + 1) // 2, 2)
    param = p
    for i in range(len(bs)):
        for j in range(20):
            if types[i][j] == 'discharge':
                if i % 2 == 0:
                    axs[i // 2, 0].plot(datas[i][j]['Time'][0][0][0], datas[i][j][param][0][0][0], label = f'{j + 1}')
                    axs[i // 2, 0].set_title(f'Battery: {bs[i]}')
                    axs[i // 2, 0].legend()
                else:
                    axs[i // 2, 1].plot(datas[i][j]['Time'][0][0][0], datas[i][j][param][0][0][0], label = f'{j + 1}')
                    axs[i // 2, 1].set_title(f'Battery: {bs[i]}')
                    axs[i // 2, 1].legend()
    for ax in axs.flat:
        ax.set(ylabel = param, xlabel = 'Time')
    fig.tight_layout(pad = 0.3)
    fig.savefig(f'PLOTS/Discharge/First/{p}_first.png')

    # Printing last cycles

    fig, axs = plt.subplots((len(bs) + 1) // 2, 2)
    for i in range(len(bs)):
        for j in range(datas[i].size - 20, datas[i].size):
            if types[i][j] == 'discharge':
                if i % 2 == 0:
                    axs[i // 2, 0].plot(datas[i][j]['Time'][0][0][0], datas[i][j][param][0][0][0], label = f'{j + 1}')
                    axs[i // 2, 0].set_title(f'Battery: {bs[i]}')
                    axs[i // 2, 0].legend()
                else:
                    axs[i // 2, 1].plot(datas[i][j]['Time'][0][0][0], datas[i][j][param][0][0][0], label = f'{j + 1}')
                    axs[i // 2, 1].set_title(f'Battery: {bs[i]}')
                    axs[i // 2, 1].legend()

    for ax in axs.flat:
        ax.set(ylabel = param, xlabel = 'Time')
    fig.tight_layout(pad = 0.3)
    fig.savefig(f'PLOTS/Discharge/Last/{p}_last.png')
```

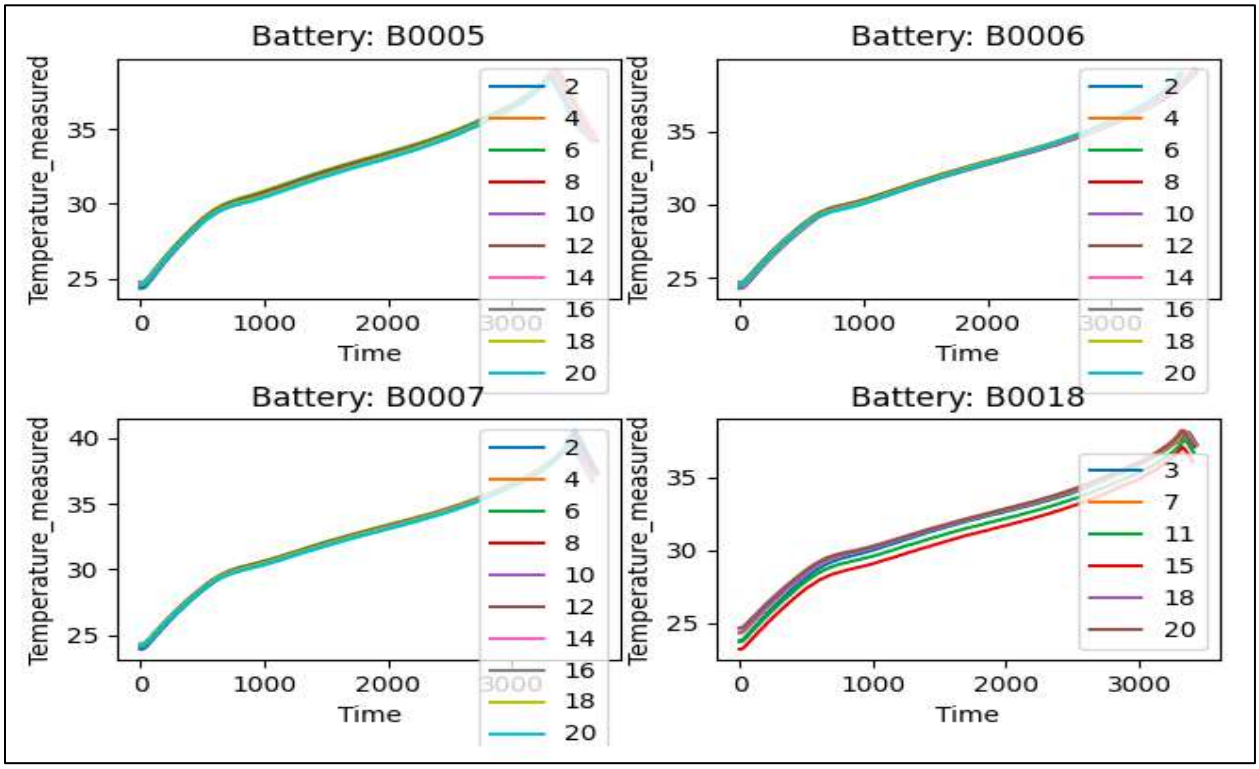
Current load first:



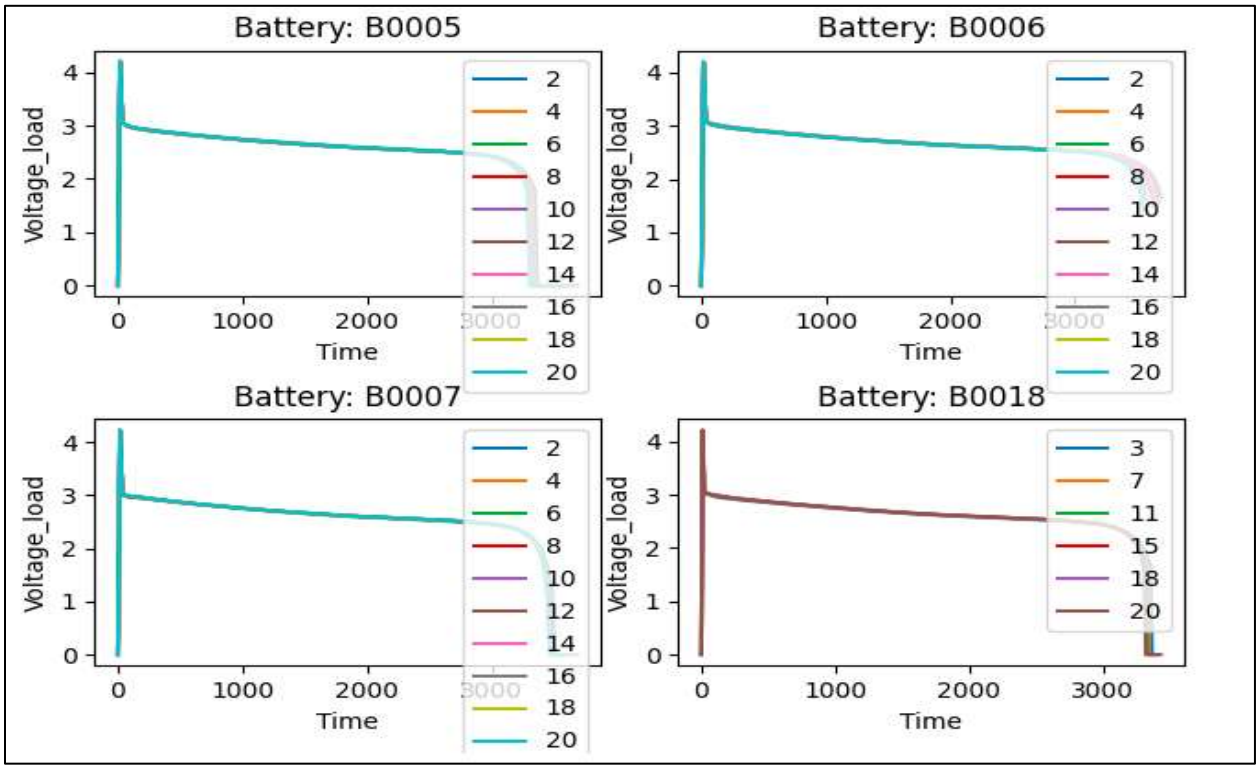
Current measured first:



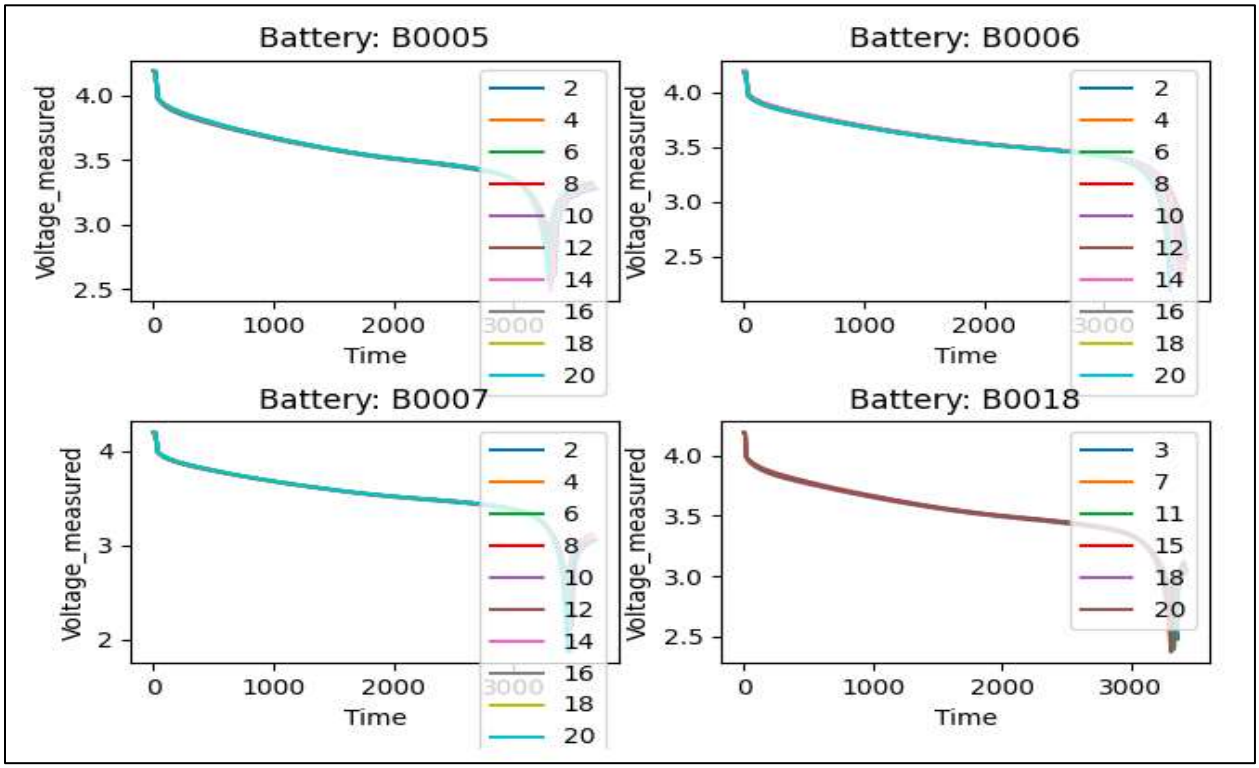
Temperature measured first:



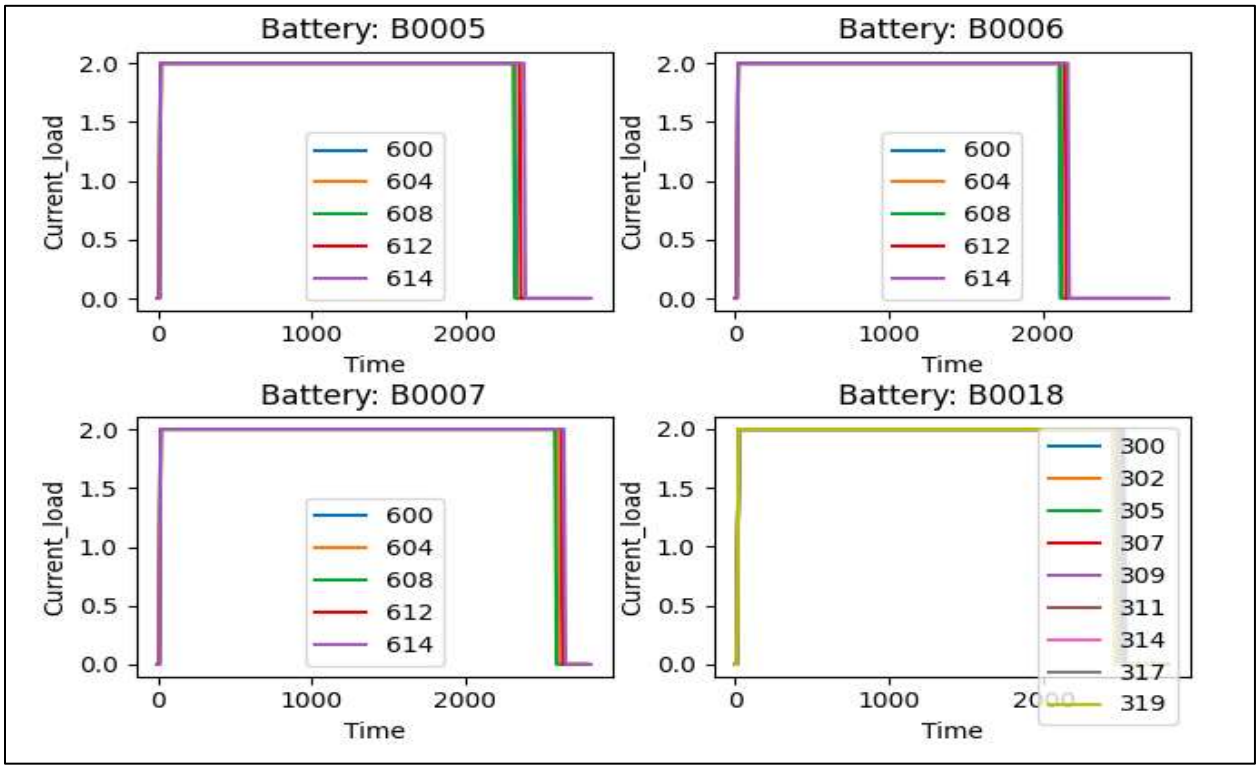
Voltage load first:



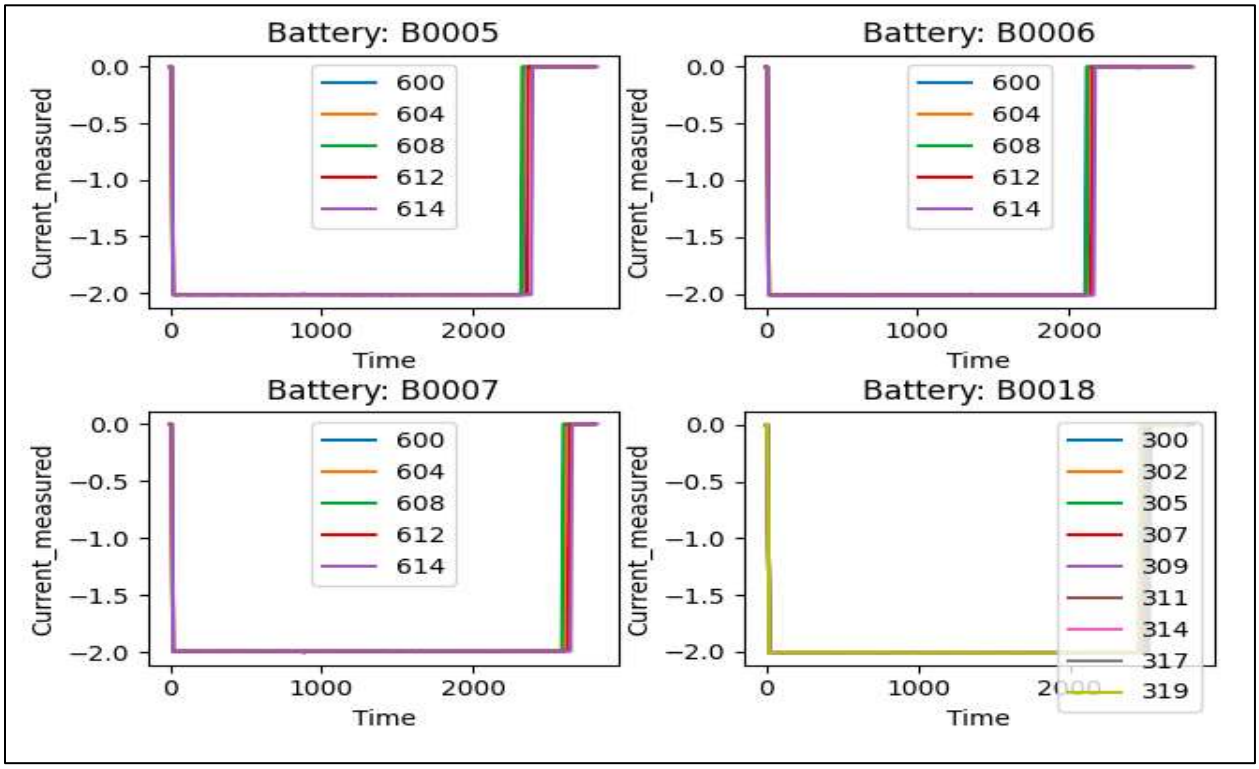
Voltage measured first:



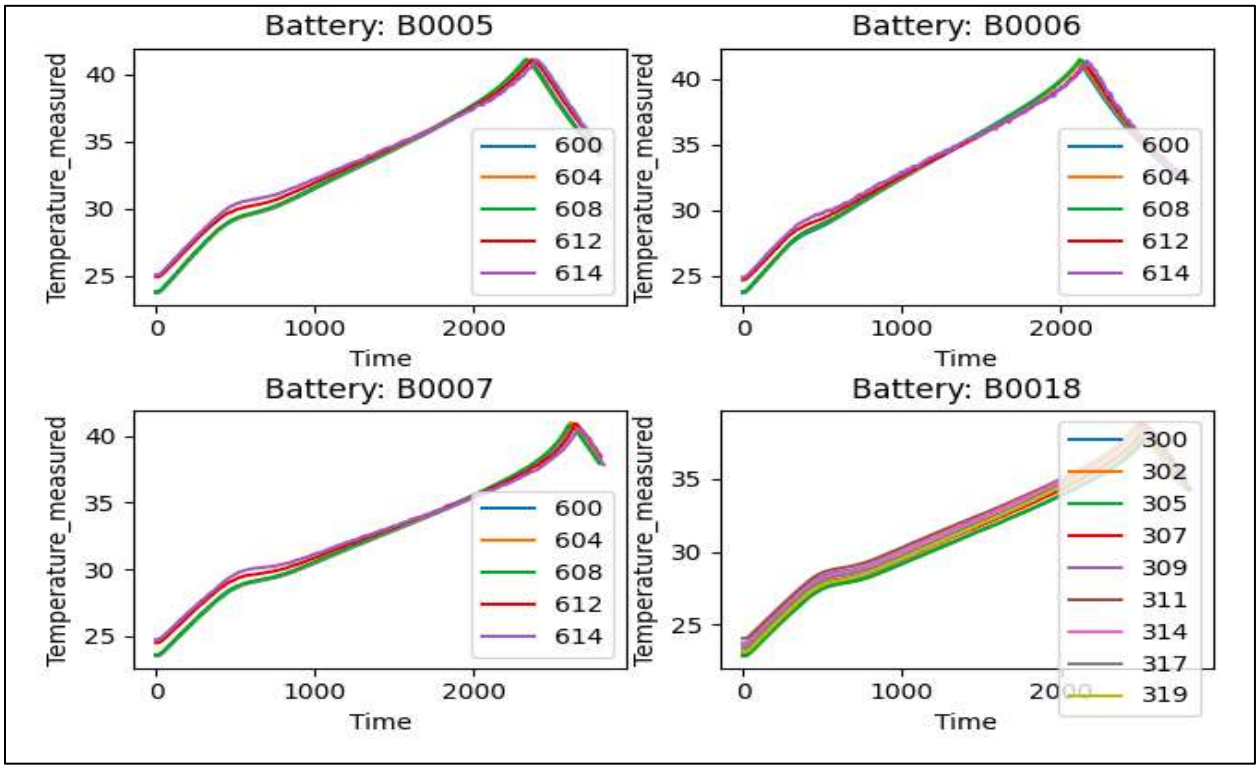
Current load last:



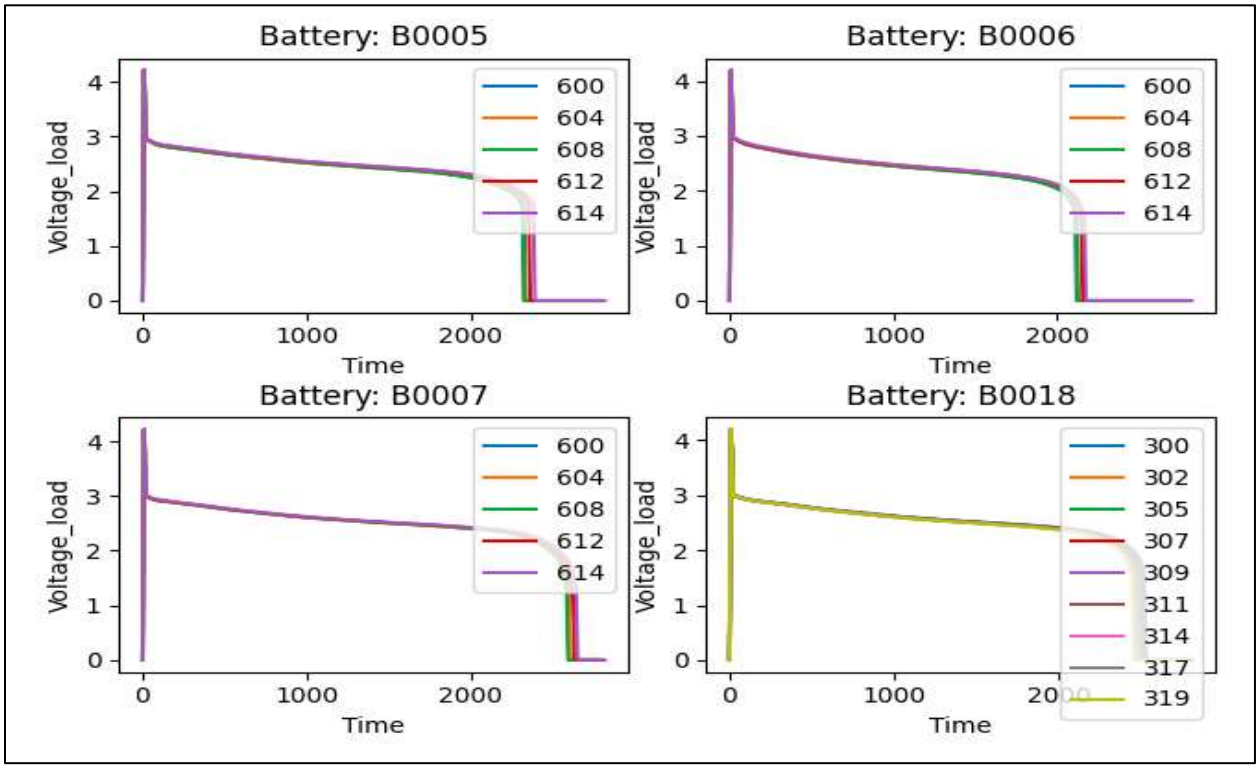
Current measured last:



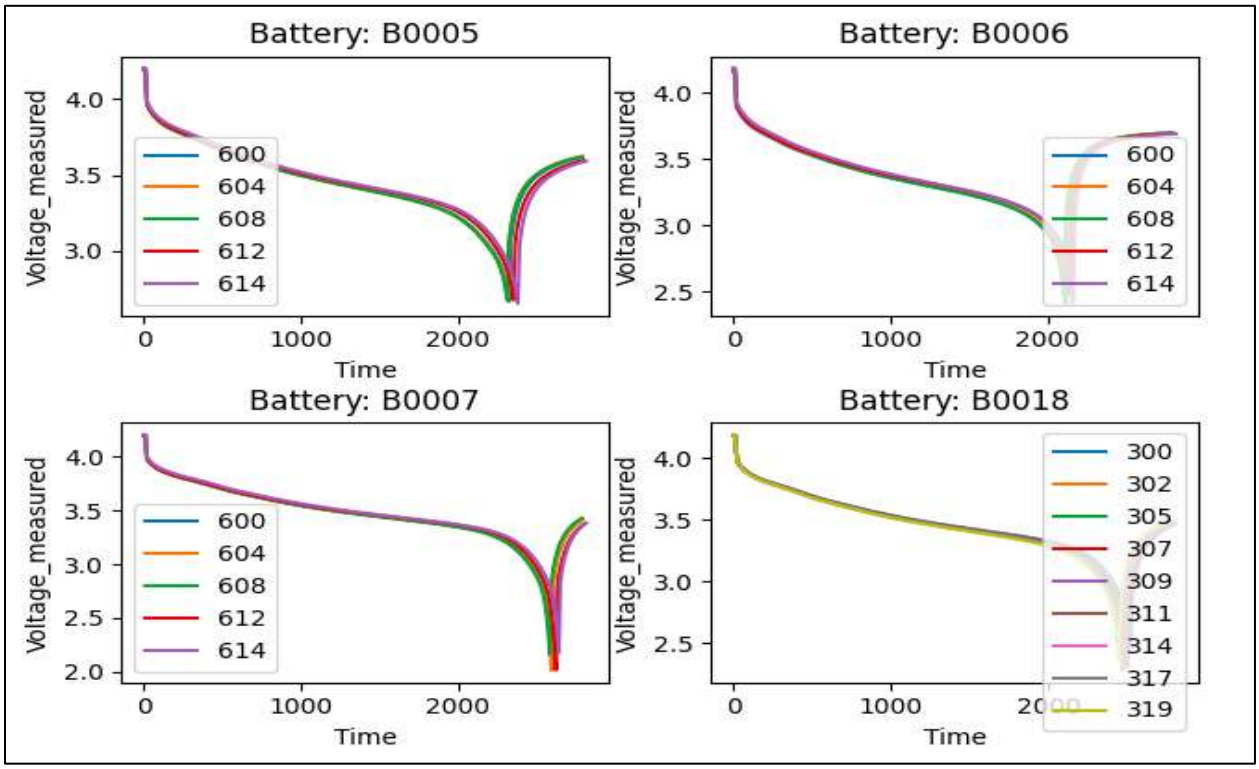
Temperature measured last:



Voltage load last:



Voltage measured last:



After plotting various parameters like Current Measured, Voltage Measured, Current Load, Voltage Load, Temperature against time, and Capacity against Cycles for four batteries, **we see a trend that the path of all the parameters is shifting as the cycle number is increasing.**

To study this **shifting trend** in more detail we plotted these parameters from **initial cycles and final cycles** and compared them against each other.

Plots for parameters vs time for the **first 20 cycles** and parameters vs time for the **last 20 cycles** to study this shifting trend in detail were also plotted.

Training the model:

In [39]:

```
from pprint import pprint

Cycles = {}
params = ['Temperature_measured', 'Voltage_measured', 'Voltage_load', 'Time']

for i in range(len(bs)):
    Cycles[bs[i]] = {}
    Cycles[bs[i]]['count'] = 168 # This is true for battery B0005, 06, 07
    for param in params:
        Cycles[bs[i]][param] = []
        for j in range(datas[i].size):
            if types[i][j] == 'discharge':
                Cycles[bs[i]][param].append(datas[i][j][param][0][0][0])

    cap = []
    for j in range(datas[i].size):
        if types[i][j] == 'discharge':
            cap.append(datas[i][j]['Capacity'][0][0][0][0])
    Cycles[bs[i]]['Capacity'] = np.array(cap)
```

In [41]:

```
# First Cycle
f = getTemperatureMeasuredCritical(Cycles[bs[0]]['Temperature_measured'][0], Cycles[bs[0]]['Time'][0])

# 100th Cycle
m = getTemperatureMeasuredCritical(Cycles[bs[0]]['Temperature_measured'][100], Cycles[bs[0]]['Time'][100])

# Last Cycle
l = getTemperatureMeasuredCritical(Cycles[bs[0]]['Temperature_measured'][167], Cycles[bs[0]]['Time'][167])

print(f'Temperature_Measured Critical points')
print(f'First Cycle:\t{f}')
print(f'100th Cycle:\t{m}')
print(f'Last Cycle:\t{l}')

## Conclusion
## !!BATTERY GET HOT QUICKER as they AGE!!
```

```
Temperature_Measured Critical points
First Cycle:    3366.781
100th Cycle:    2682.156
Last Cycle:     2393.578
```

In [42]:

```
# First Cycle
f = getVoltageMeasuredCritical(Cycles[bs[0]]['Voltage_measured'][0], Cycles[bs[0]]['Time'][0])

# 100th Cycle
m = getVoltageMeasuredCritical(Cycles[bs[0]]['Voltage_measured'][100], Cycles[bs[0]]['Time'][100])

# Last Cycle
l = getVoltageMeasuredCritical(Cycles[bs[0]]['Voltage_measured'][167], Cycles[bs[0]]['Time'][167])

print(f'Voltage_measured Critical points')
print(f'First Cycle:\t{f}')
print(f'100th Cycle:\t{m}')
print(f'Last Cycle:\t{l}')

## Conclusion
## !!VOLTAGE HOLDS FOR LESS TIME as they AGE!!
```

```
Voltage_measured Critical points
First Cycle:    3346.937
100th Cycle:    2662.828
Last Cycle:     2383.953
```

```
# First Cycle
f = getVoltageLoadCritical(Cycles[bs[0]]['Voltage_load'][0], Cycles[bs[0]]['Time'][0])

# 100th Cycle
m = getVoltageLoadCritical(Cycles[bs[0]]['Voltage_load'][100], Cycles[bs[0]]['Time'][100])

# Last Cycle
l = getVoltageLoadCritical(Cycles[bs[0]]['Voltage_load'][167], Cycles[bs[0]]['Time'][167])

print(f'Voltage_load Critical points')
print(f'First Cycle:\t{f}')
print(f'100th Cycle:\t{m}')
print(f'Last Cycle:\t{l}')

## Conclusion
## !!VOLTAGE HOLDS FOR LESS TIME as they AGE!!
```

```
Voltage_load Critical points
First Cycle:    3366.781
100th Cycle:    2672.515
Last Cycle:     2393.578
```

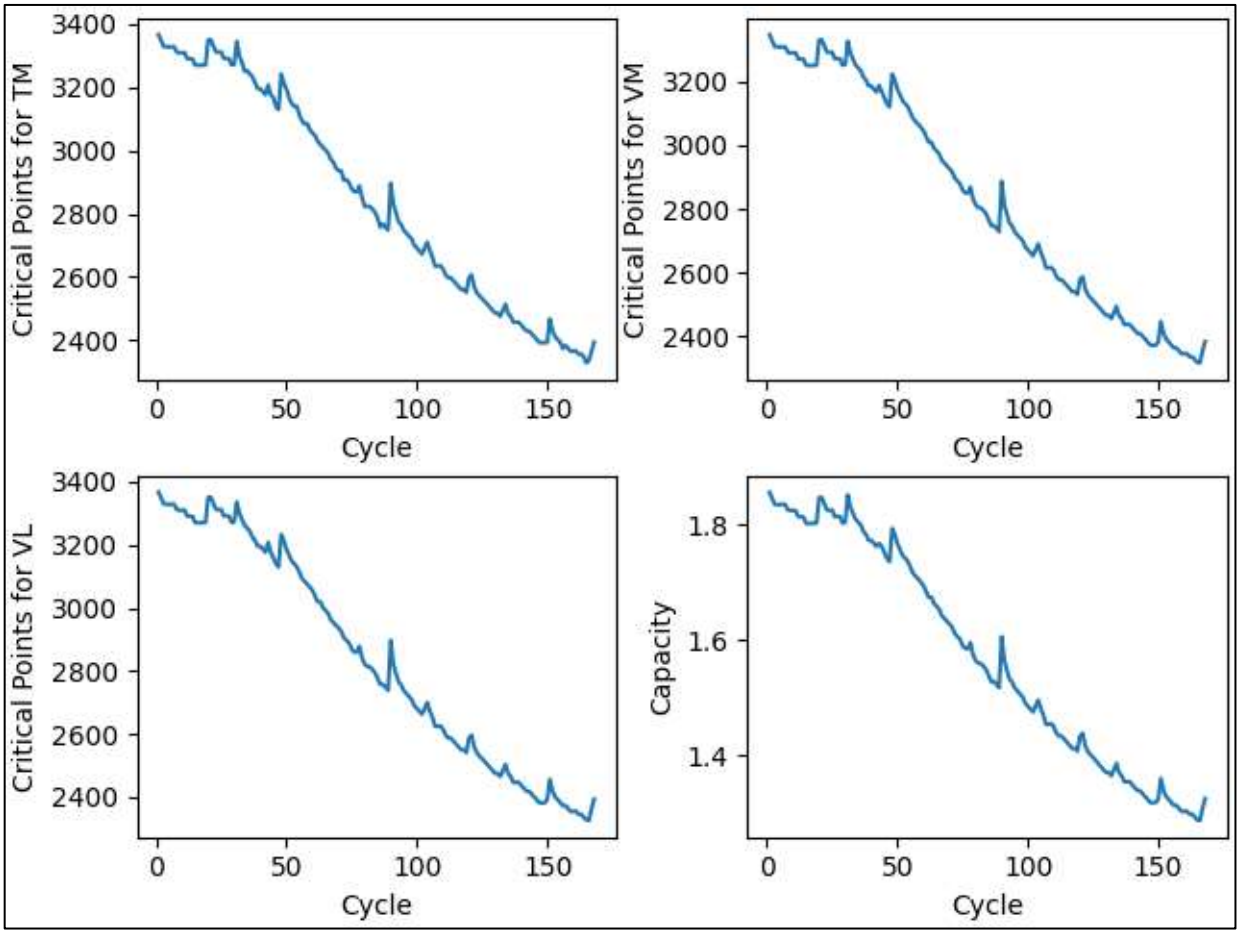
Critical Points

In the given dataset every cycle is represented by a set of arrays. Out of which Temperature, VoltageMeasured, VoltageLoad seems to best describe the cycle. These values are measured at different time points which are represented in the Time array. Rather than using the entire array for training we can extract **critical time points for each of the features** and train the model on these **critical time points.** Only using these **critical points** will reduce the training time and reduce the noise in data.

Critical points are based on the **shifting trend** discussed earlier. **The entire path shift can be thought of as a shift in its critical point**. We define critical points as follow

- **Temperature:** Time at Highest Temperature
- **Voltage Measured:** Time at lowest Voltage Measured
- **Voltage Load:** First time it drops below 1 volt after 1500 units of time

```python
## Plotting (Critical Points) v/s (Cycles)

fig, axs = plt.subplots(2, 2)
axs[0, 0].plot(range(1, len(temperature_measured) + 1), temperature_measured)
axs[0, 0].set(ylabel = 'Critical Points for TM', xlabel = 'Cycle')

axs[0, 1].plot(range(1, len(voltage_measured) + 1), voltage_measured)
axs[0, 1].set(ylabel = 'Critical Points for VM', xlabel = 'Cycle')

axs[1, 0].plot(range(1, len(voltage_load) + 1), voltage_load)
axs[1, 0].set(ylabel = 'Critical Points for VL', xlabel = 'Cycle')

axs[1, 1].plot(range(1, len(voltage_measured) + 1), capacity)
axs[1, 1].set(ylabel = 'Capacity', xlabel = 'Cycle')

fig.tight_layout(pad = 0.3)
fig.savefig(f'PLOTS/Critical_Values.png')
```



This is the plot for Battery B0005. Here cycle number is representing the age of the battery and with the increasing cycle number (age) **battery's capacity decreases** and so do the **critical values.**

Thus from the plot, it is clear that there is a **strong correlation** in the data and a simple model like **Regression** can be used for prediction.

The first regression model:

```
X = []
for i in range(Cycles[bs[0]]['count']):
    X.append(np.array([temperature_measured[i], voltage_measured[i], voltage_load[i]]))
X = np.array(X)
y = np.array(capacity)

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25, random_state = 0)

from sklearn.linear_model import LinearRegression
regressor = LinearRegression()
regressor.fit(X_train, y_train)

y_pred = regressor.predict(X_test)

diff = 0
total = 0
for i in range(len(y_test)):
    diff += abs(y_test[i] - y_pred[i])
    total += y_test[i]
diff /= len(y_test)
total /= len(y_test)
accuracy = ((total - diff) / total) * 100
print(f'Average Difference Between Predicted and Real Capacities: {diff}')
print(f'Accuracy: {accuracy}')
```

```
Average Difference Between Predicted and Real Capacities: 0.0001504458509620274
Accuracy: 99.9903768373228
```

To test the concept of critical values we first **only used one battery** i.e. B00**05** and created a dataset consisting of critical values for above mentioned 3 parameters and capacity as the label for all the discharge cycles.

We trained the model on 75% data and 25% of the data was used for testing. we obtained an accuracy of **99.99037** and an average absolute difference between predicted and the real capacity of **0.0001504**

Second regression model:

```
bs = [
    'B0005',
    'B0006',
    'B0007',
]

temperature_measured = []
voltage_measured = []
voltage_load = []
capacity = []

for b in bs:
    for c in Cycles[b]['Capacity']:
        capacity.append(c)

for _ in range(len(bs)):
    for i in range(Cycles[bs[_]]['count']):
        temperature_measured.append(getTemperatureMeasuredCritical(Cycles[bs[_]]['Temperature_measured'][i], Cycles[bs[_]]['Time'][i]))
        voltage_measured.append(getVoltageMeasuredCritical(Cycles[bs[_]]['Voltage_measured'][i], Cycles[bs[_]]['Time'][i]))
        voltage_load.append(getVoltageLoadCritical(Cycles[bs[_]]['Voltage_load'][i], Cycles[bs[_]]['Time'][i]))
```

```
X = []
for i in range(len(temperature_measured)):
    X.append(np.array([temperature_measured[i], voltage_measured[i], voltage_load[i]]))
X = np.array(X)
y = np.array(capacity)

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25, random_state = 0)

regressor = LinearRegression()
regressor.fit(X_train, y_train)

y_pred = regressor.predict(X_test)
diff = 0
total = 0
for i in range(len(y_test)):
    diff += abs(y_test[i] - y_pred[i])
    total += y_test[i]
diff /= len(y_test)
total /= len(y_test)
accuracy = ((total - diff) / total) * 100
print(f'Average Difference Between Predicted and Real Capacities: {diff}')
print(f'Accuracy: {accuracy}')
```

```
Average Difference Between Predicted and Real Capacities: 0.013404734961093146
Accuracy: 99.15812921489568
```

To test the accuracy of the model further we used **three batteries** B00**05**, **06**, **07** and created a dataset consisting of critical values for above mentioned 3 parameters and capacity as the label for all the discharge cycles and combined the data for all 3 batteries.

We trained the model on 75% data and 25% of the data was used for testing. we obtained an accuracy of **99.158129** and average absolute difference between predicted and real capacity of **0.0134047**

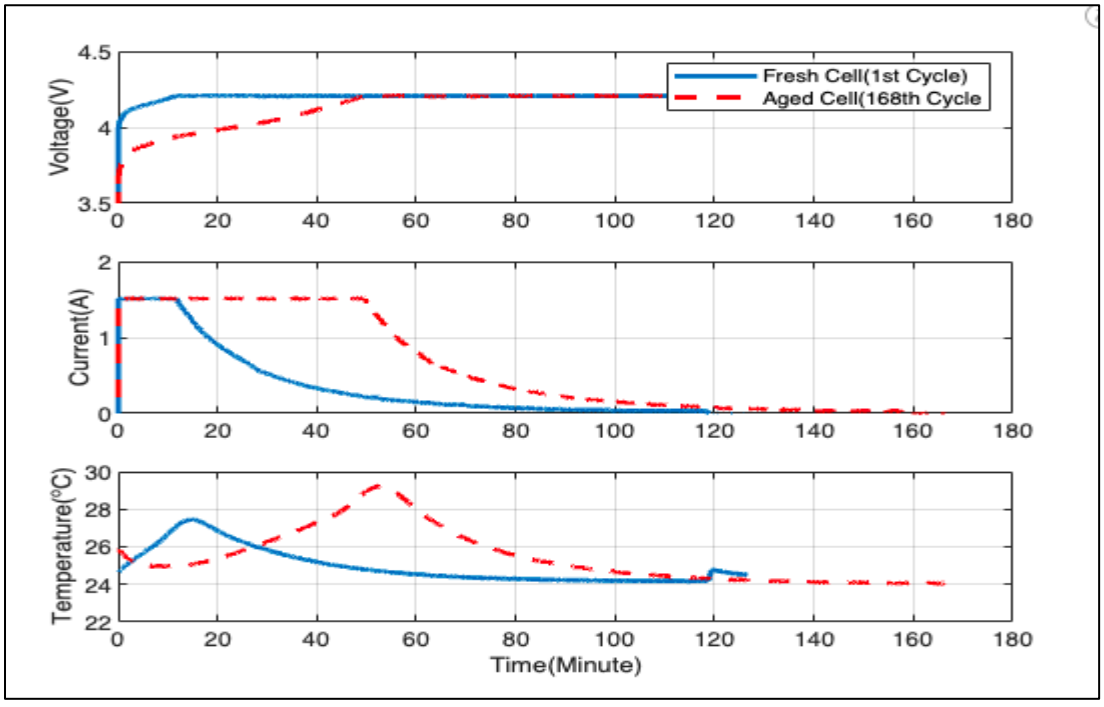
- **CNN:**

Data Analysis

We need voltage , current and temperature paths through battery aging.

```
clear;
load B0005.mat
FTime = B0005.cycle(1).data.Time/60;
FreshCell_V = B0005.cycle(1).data.Voltage_measured;
FreshCell_I = B0005.cycle(1).data.Current_measured;
FreshCell_T = B0005.cycle(1).data.Temperature_measured;

ATime = B0005.cycle(168).data.Time/60;
AgedCell_V = B0005.cycle(168).data.Voltage_measured;
AgedCell_I = B0005.cycle(168).data.Current_measured;
AgedCell_T = B0005.cycle(168).data.Temperature_measured;

figure(1)
subplot(311)
plot(FTime, FreshCell_V, 'linewidth', 2), hold on, plot(ATime, AgedCell_V, 'r--','linewidth', 2)
hold off, legend('Fresh Cell(1st Cycle)', 'Aged Cell(168th Cycle'), ylabel('Voltage(V)')
ylim([3.5 4.5]), grid on
subplot(312)
plot(FTime, FreshCell_I, 'linewidth', 2), hold on, plot(ATime, AgedCell_I, 'r--', 'linewidth', 2)
hold off, ylabel('Current(A)'), ylim([0 2]), grid on
subplot(313)
plot(FTime, FreshCell_T, 'linewidth', 2), hold on, plot(ATime, AgedCell_T, 'r--', 'linewidth', 2)
hold off, ylabel('Temperature(^oC)'), ylim([22 30]), grid on, xlabel Time(Minute)
```

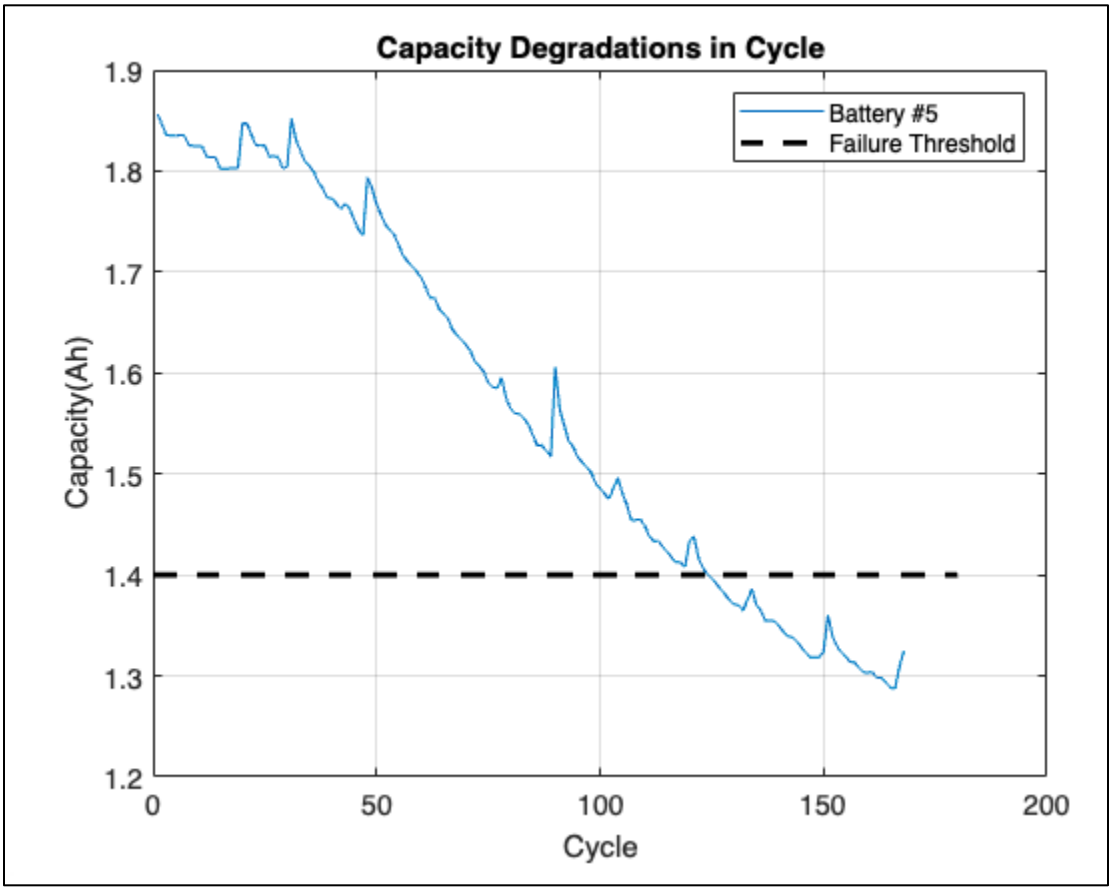
Plotting graphs of Voltage , current and temperature vs time.



Eventually, we observe the degradation characteristics per cycle as below figure.

```
cap = extract_discharge(B0005);
figure
plot(cap), hold on
plot(0:180, 1.4*ones(1, 181),'k--','LineWidth', 2)
hold off, grid on
xlabel Cycle, ylabel Capacity(Ah)
legend('Battery #5', 'Failure Threshold')
title('Capacity Degradations in Cycle')
```

Capacity degradation in cycle graph is as follows:-



Data Preprocessing

Get ready for the training. The inputs of the proposed models in are the extracted features, which are obtained by the uniform sampling of the raw battery data. Specifically, they configure the input matrix as 30-dimensional vectors by concatenating the V, I, T charging profiles, each with 10 samples. The number of samples is chosen to consider the distinct changes in time and the model complexity. In addition, we average the data over sampling interval to prevent oscillation in short time interval.

```
charInput = extract_charge_preprocessing(B0005);
```

Initial capacity for each battery data is provided as belows in the dataset:

```
InitC = 1.86;
```

For better training since it retains the original distribution of data except for a scaling factor and transforms all the data into the range of [0,1]:

```
[xB, yB, ym, yr] = minmax_norm(charInput, InitC, cap);
Train_Input = xB;
Train_Output = yB;
```

- CNN1 : 2 Convolution Layer with filter size [1, 2] and number of filter 10, 5.

```
networkNeeded = feedforwardnet(10);
networkNeeded.trainParam.epochs = 100;
[networkNeeded, tr] = train(networkNeeded, Train_Input', Train_Output', 'useparallel',  'yes');
```

Specifying the cnn model

```
layerCNN1 = [
    imageInputLayer([1, 30]);
    convolution2dLayer([1, 2], 10, 'Stride', 1);
    leakyReluLayer
    convolution2dLayer([1, 2], 5, 'Stride', 1);
    leakyReluLayer
    fullyConnectedLayer(1)
    regressionLayer();
    ];
cellx = num2cell(Train_Input', 1)';
cellx = cellfun(@transpose, cellx, 'UniformOutput', false);
cellyB = num2cell(Train_Output);
tbl = table(cellx);
tbl.cellyB = cellyB;

Traintbl = tbl(tr.trainInd, :);
valtbl = tbl(tr.valInd, :);
testtbl = tbl(tr.testInd, :);

options = trainingOptions('adam', ...
    'InitialLearnRate', 0.001, ...
    'MaxEpochs',500, ...
    'MiniBatchSize',50, ...
    'Plots','training-progress', 'ValidationData', valtbl);

netCNN1 = trainNetwork(Traintbl, layerCNN1, options);
```

```
Training on single CPU.
Initializing input data normalization.
```

Epoch	Iteration	Time Elapsed (hh:mm:ss)	Mini-batch RMSE	Validation RMSE	Mini-batch Loss	Validation Loss	Base Learning Rate
1	1	00:00:08	0.62	0.64	0.1917	0.2071	0.0010
25	50	00:00:09	0.23	0.27	0.0258	0.0374	0.0010
50	100	00:00:10	0.17	0.19	0.0141	0.0187	0.0010
75	150	00:00:10	0.12	0.14	0.0069	0.0097	0.0010
100	200	00:00:11	0.09	0.12	0.0041	0.0073	0.0010
125	250	00:00:11	0.08	0.12	0.0033	0.0073	0.0010
150	300	00:00:12	0.07	0.12	0.0028	0.0076	0.0010
175	350	00:00:13	0.07	0.13	0.0024	0.0078	0.0010
200	400	00:00:13	0.07	0.13	0.0021	0.0080	0.0010
225	450	00:00:14	0.06	0.13	0.0019	0.0081	0.0010
250	500	00:00:14	0.06	0.13	0.0017	0.0081	0.0010
275	550	00:00:15	0.05	0.13	0.0015	0.0080	0.0010
300	600	00:00:15	0.05	0.13	0.0013	0.0079	0.0010
325	650	00:00:16	0.05	0.13	0.0012	0.0078	0.0010
350	700	00:00:17	0.05	0.12	0.0011	0.0077	0.0010
375	750	00:00:17	0.05	0.12	0.0010	0.0075	0.0010
400	800	00:00:18	0.04	0.12	0.0009	0.0073	0.0010
425	850	00:00:18	0.04	0.12	0.0009	0.0071	0.0010
450	900	00:00:19	0.04	0.12	0.0008	0.0069	0.0010
475	950	00:00:19	0.04	0.12	0.0008	0.0067	0.0010
500	1000	00:00:20	0.04	0.11	0.0007	0.0065	0.0010

```
Training finished: Max epochs completed.
```

Specifying CNN model with filter size [1,2] and filter number 30,15

- CNN2 :2 Convolution Layer with filter size [1, 2] and number of filter 30, 15.

```
layerCNN2 = [
    imageInputLayer([1, 30]);
    convolution2dLayer([1, 2], 30, 'Stride', 1);
    leakyReluLayer
    convolution2dLayer([1, 2], 15, 'Stride', 1);
    leakyReluLayer
    fullyConnectedLayer(1)
    regressionLayer();
    ];
netCNN2 = trainNetwork(Traintbl, layerCNN2, options);
```

```
Training on single CPU.
Initializing input data normalization.
```

Epoch	Iteration	Time Elapsed (hh:mm:ss)	Mini-batch RMSE	Validation RMSE	Mini-batch Loss	Validation Loss	Base Learning Rate
1	1	00:00:06	0.55	0.46	0.1507	0.1063	0.0010
25	50	00:00:07	0.14	0.14	0.0101	0.0102	0.0010
50	100	00:00:07	0.10	0.14	0.0046	0.0093	0.0010
75	150	00:00:08	0.08	0.17	0.0031	0.0148	0.0010
100	200	00:00:09	0.07	0.20	0.0024	0.0193	0.0010
125	250	00:00:09	0.06	0.21	0.0018	0.0216	0.0010
150	300	00:00:10	0.05	0.21	0.0014	0.0226	0.0010
175	350	00:00:10	0.05	0.22	0.0011	0.0232	0.0010
200	400	00:00:11	0.04	0.22	0.0008	0.0237	0.0010
225	450	00:00:11	0.04	0.22	0.0007	0.0241	0.0010
250	500	00:00:12	0.03	0.22	0.0005	0.0244	0.0010
275	550	00:00:12	0.03	0.22	0.0004	0.0248	0.0010
300	600	00:00:13	0.03	0.22	0.0004	0.0253	0.0010
325	650	00:00:13	0.03	0.23	0.0003	0.0257	0.0010
350	700	00:00:14	0.02	0.23	0.0003	0.0258	0.0010
375	750	00:00:14	0.02	0.23	0.0003	0.0257	0.0010
400	800	00:00:15	0.02	0.23	0.0003	0.0256	0.0010
419	838	00:00:15	0.02		0.0003		0.0010

```
Training finished: Stopped manually.
```

Prediciton using each trained model

Make a prediction using trained models

- CNN1 : 2 Convolution Layer with filter size [1, 2] and number of filter 10, 5.

```
cellx = num2cell(Train_Input(tr.testInd, :)', 1)';
cellx = cellfun(@transpose, cellx, 'UniformOutput', false);
tbl = table(cellx);
x_4d = zeros(1, 30, 1, height(tbl));
for i = 1:height(tbl)
    x_4d(:,:,:,i) = tbl.cellx{i};
end
pCNN1 = predict(netCNN1, x_4d);
```

- CNN2 : 2 Convolution Layer with filter size [1, 2] and number of filter 30, 15.

```
pCNN2 = predict(netCNN2, x_4d);
```

Denormalization for graphical ouput. Multiplying the range of original and add minimum value.

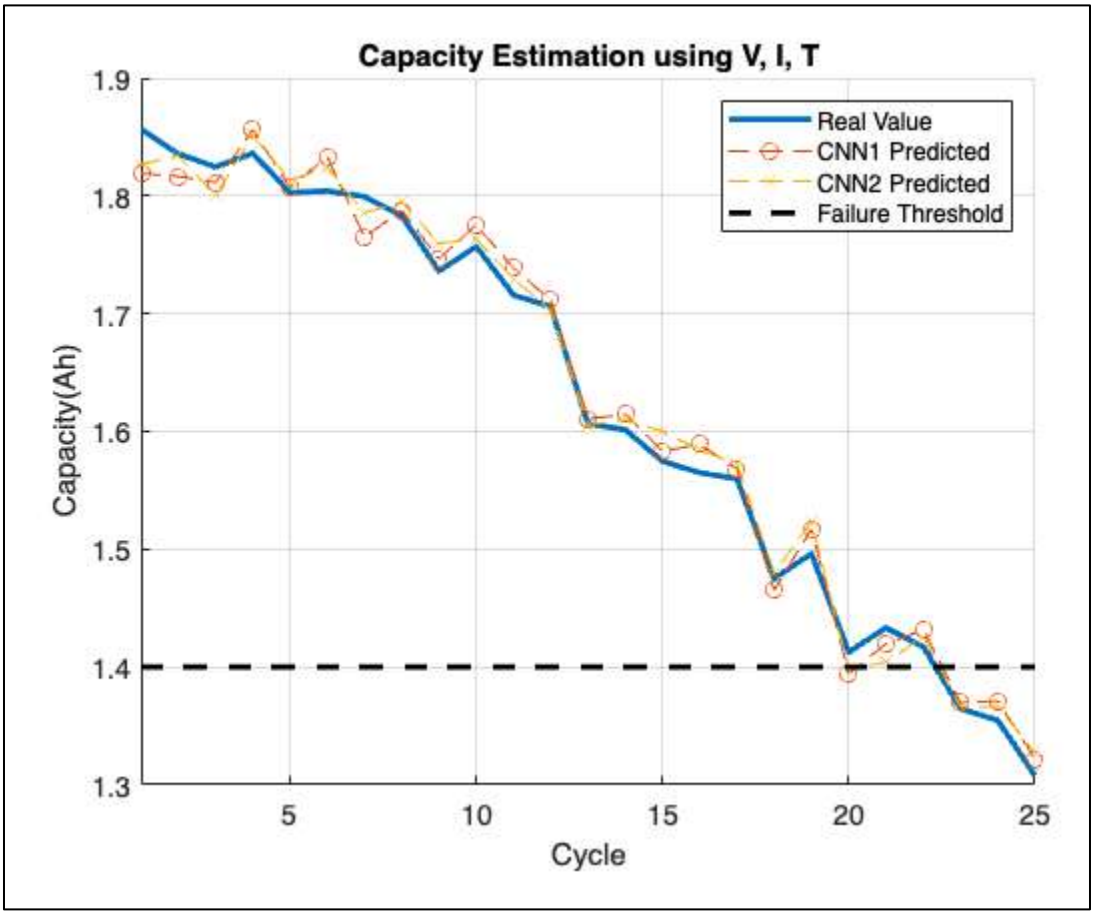
```
Train_Output = Train_Output(tr.testInd, :)*yr + ym;
pCNN1 = pCNN1*yr + ym;
pCNN2 = pCNN2*yr + ym;
```

Visualizing the prediction results:

Result visualization

Visualize the prediction result. Battery capacity estimation based multi-channel charing profiles, using Voltage, Current and temperature.

```
figure, hold on, grid on,
plot(Train_Output, 'linewidth', 2), plot(pCNN1, 'o--'), plot(pCNN2, 'x--')
plot(1:25, 1.4*ones(1, 25),'k--','LineWidth', 2), xlim([1 25])
title(['Capacity Estimation using V, I, T'])
xlabel Cycle, ylabel Capacity(Ah)
legend('Real Value', 'CNN1 Predicted', 'CNN2 Predicted', 'Failure Threshold')
```



extract_charge_preprocessing.m file:

```
function charInput = extract_charge_preprocessing(B)
bcycle = B.cycle;
for i = 1:length(bcycle)-1
    if isequal(bcycle(i).type, 'charge')
        le = mod(length(bcycle(i).data.Voltage_measured), 10);
        vTemp = bcycle(i).data.Voltage_measured(:, 1:end-le);
        vTemp = reshape(vTemp, length(vTemp)/10, []);
        vTemp = mean(vTemp);

        iTemp = bcycle(i).data.Current_measured(:, 1:end-le);
        iTemp = reshape(iTemp, length(iTemp)/10, []);
        iTemp = mean(iTemp);

        tTemp = bcycle(i).data.Temperature_measured(:, 1:end-le);
        tTemp = reshape(tTemp, length(tTemp)/10, []);
        tTemp = mean(tTemp);
        charInput(i, :) = [vTemp, iTemp, tTemp];
    end
end
charInput(~any(charInput, 2), :) = [];
```

extract_discharge.m:

```
function cap = extract_discharge(B)
bcycle = B.cycle;
for i = 1:length(bcycle)
    if isequal(bcycle(i).type, 'discharge')
        cap(i) = bcycle(i).data.Capacity;
    end
end
cap(cap==0) = [];
```

minmax_norm.m:

```
function [xData, yData, ym, yr] = minmax_norm(charInput, InitC, cap)
r = max(charInput) - min(charInput);
xData = (charInput - min(charInput))./r;
comp = length(charInput) - length(cap);
yData = [InitC*ones(comp, 1); cap'];
ym = min(yData);
yr = max(yData) - min(yData);
yData = (yData - ym)/yr;
```

- **SVM**

```
#Loading
import pandas as pd
```

```
dataset=pd.read_csv('Input n Capacity.csv')
```

```
dataset = dataset.drop(labels=['SampleId'], axis=1)
data = dataset[~dataset.isin(['?'])]
data = data.dropna(axis=0)
data.shape
```

```
(636, 6)
```

```
data.size
```

```
3816
```

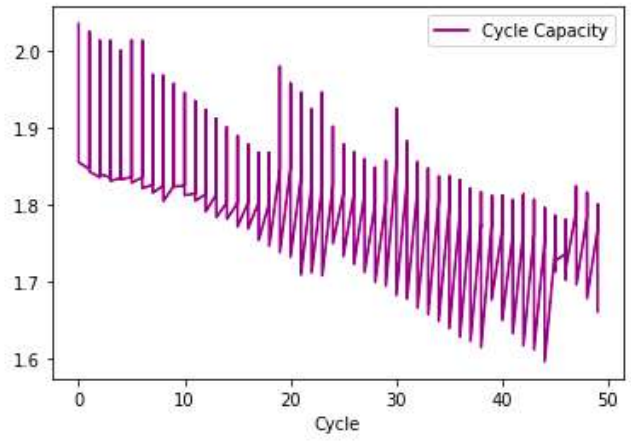
```
data.count()
```

```
Cycle                    636
Time Measured(Sec)       636
Voltage Measured(V)      636
Current Measured         636
Temperature Measured     636
Capacity(Ah)             636
dtype: int64
```

Plotting the distribution (cycle capacity):

```
#Distribution
CycleC = data[1:200]
CycleC.plot( x='Cycle', y='Capacity(Ah)', color='purple', label='Cycle Capacity')
```

```
<AxesSubplot:xlabel='Cycle'>
```



```
import numpy as np
X = np.array(data.iloc[:,0:5].values)
y = np.array(data.iloc[:,5].values)
```

```
array([[ 0.00000000e+00,  2.03533759e+00,  2.47576776e+00,
        -2.00943589e+00,  3.91629865e+01,  3.69023400e+03],
       [ 0.00000000e+00,  1.89105229e+00,  3.06211271e+00,
        -1.43329900e-03,  3.73384785e+01,  3.69023400e+03],
       [ 0.00000000e+00,  1.85500452e+00,  3.05323034e+00,
        -2.43341500e-03,  3.72056713e+01,  3.43489100e+03],
       [ 1.00000000e+00,  1.84632725e+00,  3.30024489e+00,
        -4.47553000e-04,  3.43921366e+01,  3.67234400e+03],
       [ 1.00000000e+00,  2.02514025e+00,  2.35152551e+00,
        -2.01037486e+00,  3.92462027e+01,  3.67234400e+03],
       [ 1.00000000e+00,  1.88063703e+00,  3.07922624e+00,
        -3.23038500e-03,  3.71617386e+01,  3.67234400e+03],
       [ 1.00000000e+00,  1.84319553e+00,  3.08820017e+00,
        -9.10648000e-04,  3.71554752e+01,  3.42548500e+03],
       [ 2.00000000e+00,  1.83534919e+00,  3.32745101e+00,
         1.02601900e-03,  3.42327787e+01,  3.65164100e+03],
       [ 2.00000000e+00,  2.01332637e+00,  2.44047971e+00,
        -2.00855889e+00,  3.89992025e+01,  3.65164100e+03]])
```

In [108]:
```
#Modeling
from sklearn import svm
parameters = [{'C': [0.001, 0.1 ,1,5], 'kernel':['linear']},
              {'C': [0.001, 0.1, 1,5], 'kernel':['rbf'], 'gamma':[0.01, 0.05]}]
classifier = svm.SVC(gamma=0.001, C= 0.001, kernel='linear')
#classifier = SVC(kernel='rbf', random_state = 1)
classifier
```

Out[108]:
```
SVC(C=0.001, gamma=0.001, kernel='linear')
```

In [116]:
```
import numpy as np
from sklearn                          import metrics, svm
from sklearn import preprocessing
from sklearn import utils

lab_enc = preprocessing.LabelEncoder()
y_encoded = lab_enc.fit_transform(y_train)
print(y_encoded)
print(utils.multiclass.type_of_target(y_train))
print(utils.multiclass.type_of_target(y_train.astype('int')))
print(utils.multiclass.type_of_target(y_encoded))
y_encoded.shape
```

```
[276 438 153 480 109   4 414  35 166 181 369 502 396  81 437 196 278 176
 381 395  45 223 284 456 351 115  76 318 406 266 312 475 477  60  56 382
 430 158 144  65 325 385 378 148 410 404 195 192 393  37 355 472 124 457
 252 304 345 446 358 441 170 128 163 132 187 341   8 237 174 449 160 292
 337 246 398 392  27 423 469 279 493  80 258 445 133 236 425 380  16 331
 330 302 379 303 420 239 204 462 467 141 336  41   3 261  22  28 190  95
 316   7  89 463 220  97 435 453 503 110 360  39 432 157 361 168  40 256
  74 338 507 329 171 487  66 263 229 226 307 495 159 300 388 215 496 149
 383 483 344 240 418 227 320 233 180 257 459 107 428 324 260 287 444 367
 468 327 374  25 172 203  17 268 415 207 208 295  42  14  70 426 416 315
 376 386   0 357 145 112  15 401 362 269 108 177 433 272  85 288  93 339
 334  20 490 403 412 442  62 232 184 249 505  36  52 377 470  83 230 306
 134 116 104  87 285 238  99 186 471 117 308  51 118  18 452 373 282 387
 139  73 135 136 305 281 291 460 424 440 264 119 165 156 346  34 102 474
  72 194 368 206 372 169 501  67 154 443 273 137 244   1 494 155 167 409
 321  44 465 326 147  94 162 222  71  12  75 213  82 394  30 274  86 340
  32 413 106 359 103 175 179 364 289  24 439  79   6 419 161 142  88 189
  21 407 391 277  98 234  23 164 350 250 347 366 201 497 294 451 235 310
  33 436 342 243 473 254 253 434 185 265 408   9 314 309 399 498  63 489
  13 492  68 241 476 354 199 197 183 448 202 500 332 247 319 299  96 375
 140 328 111 275 293 221  59 138 191  11 283 290 479 311  38  92 488  90
 429  91 499 217 200 298  84 506 188 242 431 122  53 384  58  47 245 218
 211 224  26 478 447 365 123 270 214 491 146 417 280 173 114 101  57 335
 130 267   2  54 482 296 427 205 100 322  55 129 323 485 504 481  61 125
 151 458 216 228 113 349 120 363  50 313 397 353 389 248  10  78 348  43
 225 402  31 210 150 178 422  77 370 450 301 126 352 466 333 400  19 251
 343 127 286  69 209  64 371 255 271 193 182 297  49 484 198   5 405 231
  48 259 390 212 356 455 105  46 464 317 421 219  29 454 461 152 131 143
 262 121 411 486]
```

Printing the accuracy:

```
Accuracy: 92.3621986207
```

Printing y_test:

```
In [119]: y_test

Out[119]: array([1.68882112, 1.74965007, 1.35326909, 1.41257879, 1.66075914,
                 1.83682843, 1.81320418, 1.7060145 , 1.65742431, 1.5145493 ,
                 1.44179059, 1.84852529, 1.36273718, 1.60151422, 1.8060549 ,
                 1.57521855, 1.31320206, 1.4478661 , 1.76231507, 1.70810865,
                 1.86003391, 1.48332385, 1.85180255, 1.8451998 , 1.87827837,
                 1.76047124, 1.37618266, 1.45192377, 1.69357162, 1.56498205,
                 1.83070385, 1.29807351, 1.51394706, 1.84781729, 1.84741731,
                 1.43867094, 1.83534919, 1.92424611, 1.91188993, 1.72856423,
                 1.38931712, 1.75467665, 1.77026297, 1.39453438, 1.7686304 ,
                 1.81517001, 1.60656314, 1.81570251, 1.20561598, 1.47711553,
                 1.15879749, 1.3231709 , 1.73151667, 1.25343543, 1.6058189 ,
                 1.31829304, 1.81396939, 1.41657818, 1.30901536, 1.83960184,
                 1.96816618, 1.20089363, 1.45669527, 1.40759837, 1.58762731,
                 1.67605161, 1.847026  , 1.52225959, 1.63785784, 1.59041024,
                 1.64822416, 1.5641733 , 1.5065639 , 1.41545478, 1.42608775,
                 1.59580651, 1.82371946, 1.74621974, 1.47520959, 1.54501875,
                 1.31564853, 1.18517925, 1.35470392, 1.57997371, 1.45390123,
                 1.67456925, 1.45162927, 1.585789  , 1.80277763, 1.66743672,
                 1.55117119, 1.74184961, 1.80306831, 1.40120378, 1.30023574,
                 1.42070062, 1.66641093, 1.81476919, 1.43624562, 1.62775289,
                 1.349315  , 1.62919994, 1.54639024, 1.43121075, 1.58660118,
                 1.66226626, 1.54387715, 1.85500452, 1.37539202, 1.37871902,
                 1.54459401, 1.84837895, 1.63157008, 1.53430867, 1.43344543,
                 1.69759542, 1.38611152, 1.34743288, 1.64930064, 1.68307444,
                 1.3052162 , 1.55561827, 1.59546387, 1.5118976 , 1.44713797,
                 1.43629451, 1.61090284, 1.44167419])
```

Printing y_pred:

```
In [120]: Y_pred

Out[120]: array([278, 367,  65,  65, 345, 445, 379, 316, 317, 242,  47, 465,  76,
                 222, 411, 216,  32, 169, 363, 365, 460, 206, 492, 406, 404, 384,
                  70, 237, 318, 264, 485,  27, 224, 404, 498, 116, 443, 473, 491,
                 364, 116, 379, 385, 161, 373, 410, 300, 422,   7, 188,   1,  35,
                 344,  42, 278,  12, 454, 117,   3, 436, 497,  29, 163,  65, 272,
                 319, 481, 220, 306, 227, 309, 271, 165, 127, 134, 282, 398, 369,
                 128, 172,  71,   4,  61, 274, 116, 283, 249, 273, 466, 300, 252,
                 348, 466,  64,  28, 200, 321, 467, 151, 235,  24, 336, 262, 199,
                 277, 300, 170, 447,  38, 125, 161, 465, 269, 255,  90, 361, 212,
                 212, 309, 330,  60, 251, 291, 176, 153,  12, 296, 228], dtype=int64)
```

Thus, this model outputs an array of values for y_test and y_pred and it also plots the cycle capacity distribution graph.

Accuracy: 92.3621986207

Results

We have used ANN model for predicting the RUL of Lithium-ion batteries which has an accuracy of 99%. We dint implement modified version of Bat-PF in the base code as the base code of bat pf itself wasn't working . Hence we decided to code the full Ann model from scratch and optimized it to get a 99 percent accuracy. From the above comparison results and our implementation of ANN , we conclude that , ANN is a good choice for RUL Prediction as it has various advantages like Problems represented by attribute-value pairs , Output may be discrete-valued, real-valued, or a vector of several real or discrete-valued attributes , Robust to noise in the training data. Fast evaluation of the learned target function and Able to bear long training times . Based on the previous summary, the advantages and disadvantages of the proposed methods are compared. SVM has satisfactory performance in nonlinear and high-dimensional models, can deal with local minima and small sample sizes, and has a short calculation time. However, it cannot express uncertainty due to its difficulty with calculating kernel and regularization parameters. Regression models cannot work properly if the input data has errors (that is poor quality data). If the data preprocessing is not performed well to remove missing values or redundant data or outliers or imbalanced data distribution, the validity of the regression model suffers. While regression is a useful tool for analysis, it does have its disadvantages, including its sensitivity to outliers and more. CNN comes ahead strong, compared to its predecessors, in that it automatically detects the important features without any human supervision. However, a Convolutional neural network is significantly slower due to an operation such as maxpool. If the CNN has several layers then the training process takes a lot of time if the computer doesn't consist of a good GPU. Artificial neural networks are algorithms that can be used to perform nonlinear statistical modeling and provide a new alternative to logistic regression, the most suitable in our case for developing predictive models for RUL prediction. Neural networks offer a number of advantages, including requiring less formal

statistical training, ability to implicitly detect complex nonlinear relationships between dependent and independent variables, ability to detect all possible interactions between predictor variables, and the availability of multiple training algorithms.

Conclusion

The proposed ANN method is validated using the condition monitoring data collected by Prognostics Center of Excellence at Ames Research Center, NASA. Experiment results show that the proposed ANN method can produce satisfactory RUL prediction results, which will assist the condition based maintenance optimization. A comparative study is performed between the proposed ANN method and other methods, and the results demonstrate the clear advantage of the proposed approach in achieving more accurate predictions. The accuracy of RUL prediction of lithium-ion batteries depends on both the degradation model and the online model updating method.

We achieved a very high accuracy when we used the ANN model of 99% . However when we were considering other activation functions in ann, the accuracy was low . Hence we used the sigma , relu and tanh activation functions.Our model working on accuracy with 99% for one independent variable were considered individually.In this project, we propose a novel intelligent RUL prediction approach using the ANN degradation model and sgd optimizer. This work developed and evaluated an ANN-based intelligent algorithm for RUL prediction of lithium-ion batteries under various training datasets. The battery dataset for training the model was acquired from the NASA Prognostics Centre of Excellence Data Repository. Compared to the other methodology employing the SCI profile, the MCI profile was applied to achieve better results in predicting the RUL under different combinations of battery datasets. Various significant parameters such as voltage, current, and temperature samples were obtained from the charging cycles for the accurate prediction of the RUL of lithium-ion batteries. Due to the employment of a systematic sampling approach for critical data extraction, it is easy to reconstitute the predicted capacity curve while training with various combinations of battery datasets.Comparing with the conventional empirical model and CNN models as well as with regression and

SVM models , we conclude that the proposed NN model has no requirement on the degradation pattern and can be adapted to various dynamic trends, which achieves better performance. On the other hand, by integrating the optimization algorithm with the standard PF, we introduce a promising method to online update the degradation model parameters and estimate the RUL. Benefiting from the unique mechanism of the algorithm, which utilizes the newly capacity data to improve the PF particle distribution by moving the particles to high likelihood locations, our proposed method avoids the resampling process and thus prevents the particle degeneracy and impoverishment problem in standard PF method. Furthermore, we quantitatively evaluate the proposed NN model and sgd optimizer method using two representative datasets, which are collected under different cycling conditions. The experimental results indicate that our method can better model the capacity degradation trend and obtain a higher RUL prediction accuracy (small prediction error and narrower pdf width) compared with conventional methods. In future work, we are interested in extending our prediction method to more practical based applications. On the other hand, by exploring the relationship between the degradation model and the operation conditions, such as the working temperature and current, we also plan to build a general degradation model.

Future Scope

In future work, we are interested in extending our prediction method to more practical based applications. On the other hand, by exploring the relationship between the degradation model and the operation conditions, such as the working temperature and current, we also plan to build a general degradation model. In the future, authors will investigate the accuracy of the multi-step approach for RUL prediction of batteries cycled using continuously changing load profiles, like drive cycles. Being a data-driven approach, accuracy of the multi-step model depends on the data availability in the testing region. In the future, active online learning algorithms can be investigated such that the multi-step approach can adaptively learn online whenever new data is available.

References

- Multi-Channel Profile Based Artificial Neural Network Approach for Remaining Useful Life Prediction of Electric Vehicle Lithium-Ion Batteries
- Remaining Useful Life Prediction of Lithium-Ion Batteries Using Neural Network and Bat-Based Particle Filter https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8698866
- Cycling Lifetime Prediction Model for Lithium-ion Batteries Based on Artificial Neural Networks.Mohsen Vatani, Preben J.S. Vie, and Oystein Ulleberg Institute for Energy Technology (IFE)
- Novel Statistical Analysis Approach for Remaining Useful Life Prediction of Lithium-Ion Battery.
- C. Hendricks, N. Williard, S. Mathew, and M. Pecht, ‘‘A failure modes, mechanisms, and effects analysis (FMMEA) of lithium-ion batteries,’’ J. Power Sources, vol. 297, pp. 113–120, Nov. 2015.
- K. Javed, R. Gouriveau, and N. Zerhouni, ‘‘State of the art and taxonomy of prognostics approaches, trends of prognostics applications and open issues towards maturity at different technology readiness levels,’’ Mech. Syst. Signal Process., vol. 94, pp. 214–236, Sep. 2017.
- Y. Zhang, R. Xiong, H. He, and M. Pecht, ‘‘Validation and verification of a hybrid method for remaining useful life prediction of lithium-ion batteries,’’ J. Cleaner Prod., vol. 212, pp. 240–249, Mar. 2019.