State of Charge Estimation of Li-ion Battery

Thesis submitted in partial fulfillment for the Award of

B.TECH DEGREE

in

ELECTRICAL ENGINEERING

by

Chepuri Vishwas 19085025

Koneru Saketh 19085047

Shaik Asif Ahmad 19085082

DEPARTMENT OF ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY

(BANARAS HINDU UNIVERSITY)

VARANASI - 221005

CERTIFICATE

It is certified that the work contained in the thesis titled **State of Charge Estimation of Li-ion Battery** has been carried out under my/our supervision and that this work has not been submitted elsewhere for a degree.

It is further certified that the student has fulfilled all the requirements of Comprehensive Examination, Candidacy and SOTA for the award of **B.TECH Degree**.

Supervisor
Dr. Sandip Ghosh
Department of Electrical Engineering
Indian Institute Of Technology
(Banaras Hindu University)
Varanasi – 221 005

DECLARATION BY THE CANDIDATES

We, **Chepuri Vishwas, Koneru Saketh, Shaik Asif Ahmad**, certify that the work embodied in this thesis is my own bona fide work and carried out by me under the supervision of **Dr. Sandip Ghosh** from **January 2022** to **December 2022**, at the **Department of Electrical Engineering**, Indian Institute of Technology (Banaras Hindu University), Varanasi. The matter embodied in this thesis has not been submitted for the award of any other degree/diploma. We declare that I have faithfully acknowledged and given credits to the research workers wherever their works have been cited in my work in this thesis. We further declare that we have not willfully copied any other's work, paragraphs, text, data, results, *etc.*, reported in journals, books, magazines, reports dissertations, theses, *etc.*, or available at websites and have not included them in this thesis and have not cited as my own work.

Date: December 5, 2022 Signature of the Student
Place: Varanasi, India

CERTIFICATE BY THE SUPERVISOR

It is certified that the above statement made by the student is correct to the best of my/our knowledge.

Supervisor
Dr. Sandip Ghosh
Department of Electrical Engineering
Indian Institute Of Technology
(Banaras Hindu University)
Varanasi – 221 005

Signature of Head of Department

ACKNOWLEDGEMENT

We would like to thank our supervisor Dr. Sandip Ghosh sir for the continuous support and advice throughout the project. His guidance has helped us all the time and we have learned a lot through this project.

Contents

1. **Problem Statement**

1.1 State of Charge

1.2 Coulomb Counting

2. **Data Sources Used**

3. **Various Estimation Methods**

3.1 Extended Kalman Filter

3.2 Feedforward Neural Network

3.3 Decision Tree Ensemble

4. **Experiments and Results**

5. **References**

1. Problem Statement

The automotive industry is currently experiencing a paradigm shift from conventional, diesel and gasoline propelled vehicles into the second generation hybrid and electric vehicles. Since the battery pack represents the most important and expensive component in the electric vehicle powertrain, extensive monitoring and control is required. Therefore, extensive research is being conducted in the field of electric vehicle battery condition monitoring and control. At present, the main challenge for the wide acceptance of electric vehicles lies mainly in their initial cost due to using battery systems, which represent the most expensive components.

Manufacturers tend to over design the battery system in Electric Vehicles to compensate for any uncertainties in battery system modeling. Therefore improving battery system modeling accuracy results in smaller pack size, which in turn directly lowers the system's cost and consequently, the vehicle's cost. The Battery Management System (BMS) is responsible for accurately estimating, in real-time, the stored energy in the vehicle batteries. This is a relatively complex task due to many aspects. First, there is no direct measurement method that can directly measure the stored energy in the battery pack. In this project, we used various estimation algorithms for Lithium-Ion (Li-Ion) battery state-of-charge (SOC) estimation.

1.1 State of Charge

State of Charge is defined as the ratio of the remaining charge in the battery and the maximum charge that can be delivered by the battery.

$$SoC(t) = \frac{Q(t)}{Q_n}$$

State Of Charge estimation is a fundamental challenge for battery use. The SOC of a battery, which is used to describe its remaining capacity, is a very important parameter for a control strategy. As the SOC is an important parameter, which reflects the battery performance, so accurate estimation of the SOC can not only protect battery, prevent over discharge, and improve the battery life but also allow the application to make rational control strategies to save energy.

1.2 Coulomb Counting

SOC Estimation by Coulomb Counting is based on the measurement of the current and integration of that current over time. Coulomb counting gives a relative change in SoC and not an absolute SoC. If you measure the current over a given time step you have a measure of the number of Ah that have left or been received by the battery.

$$SoC(t) = SoC(t-1) + \frac{I(t)}{Q_n}\Delta t$$

Drawback of this method is that due to the integration, any measurement error in current accumulates over time leading to inaccurate results.

2. Data Sources Used

To train and run estimation algorithms used in this project, one needs battery charge and discharge data. So an open source dataset was used in this project. That open source data consists of charge, discharge data and also discharge data under various drive cycles like UDDS, HWFET, LA92, US06 under a temperature controlled chamber at various ambient temperatures like 40ºC, 25ºC, 10ºC, 0ºC, -10ºC, and -20ºC on a new 5Ah Turnigy Graphene cell. Data pertaining to each charge, discharge or drive cycles consists of Measured Temperature, Measured SOC values, Measured Terminal Voltage and Measured Current.

3. Various Estimation Methods

3.1 Extended Kalman Filter

Kalman Filtering is an algorithm that uses a series of measurements observed over time, including statistical noises and other inaccuracies, and produces estimates of unknown variables that tend to be more accurate than those based on a single measurement alone, by estimating a joint probability distribution over the variables for each timeframe.

Essentially, by converging the error of estimated output w.r.t real output to zero, we converge the value of the estimated state variable to the actual value. This is just like a feedback control system. Assuming a Gaussian distribution of the probability density function of varied covariances for the estimated values of variables, we can use all of them to get a much more nuanced and accurate value of desired variables. The Kalman Filter can be used to model linear systems but cannot be used to model non-linear systems. Whereas the Extended Kalman Filter (EKF) can be used when a system is non-linear but can be linearized about an estimate of the current mean and current co-variance.

EKF linearizes the battery model using partial derivatives and first order Taylor series expansion. The state-space model is linearized at each time instance, which compares the predicted value with its measured battery terminal voltage to correct the estimation parameters for SOC. However, if the system is highly non-linear, linearization error may occur due to the lack of accuracy in the first order Taylor series under a highly non-linear condition.

The approximation is better if:

1) The covariance of the estimates is lower.

2) Less divergence between the linearised function and the original function.

Essentially, EKF is a two-step prediction-correction algorithm where first the variables are assigned a priori values through prediction and then are updated using corrections giving us the a posteriori values which will now be used in the next iteration.

EKF has three variables P, Q and R which are:

P = Covariance of a priori predicted state estimate

Q = Covariance of Gaussian distribution of process noise

R = Covariance of Gaussian distribution of measurement/output noise

- Equations for prediction step(a priori values of state variable and error covariance):

$$\widehat{x}_{k+1|k} = A\widehat{x}_{k|k} + Bu_k$$

$$\widehat{P}_{k+1|k} = A\widehat{P}_{k|k}A^T + Q_k$$

- Equations for computing Kalman gain and updating the previous measurements:

$$K_{k+1} = P_{k+1|k}C^T(CP_{k+1|k}C^T + R_{k+1})^-1$$

$$\widehat{x}_{k+1|k+1} = \widehat{x}_{k+1|k} + K_{k+1}(z_{k+1} - C\widehat{x}_{k+1|k})$$

$$\widehat{P}_{k+1|k+1} = (1 - K_{k+1}C)P_{k+1|k}$$

- Equation for Terminal Voltage:

$$V_t = V_{OC} - V_1 - V_2 - iR_0$$

3.2 Feedforward Neural Networks

Artificial neural networks (ANNs) are universal approximators that can model any nonlinear function with desired accuracies. The ANN is a mathematical model that consists of many nonlinear artificial neurons running in parallel, which may be created as one layer or multiple layers. It is widely used in function fitting, data clustering and pattern recognition [11]. Feedforward neural networks are one of the most commonly used networks that have good performance prediction.

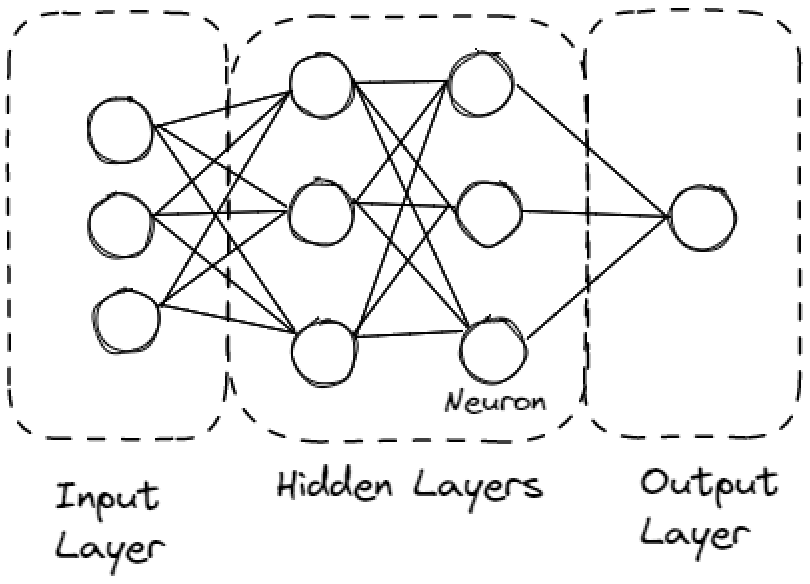

Fig. General Architecture of FNN

In general, an artificial neural network contains three main layers; an input layer with nodes to symbolize the input variables, one or more hidden layers with nodes to reproduce the nonlinearity between the system input and output and an output layer to represent the system output variable. In this work, we applied feed forward NN because of its easy implementation and ability for nonlinear simulation.

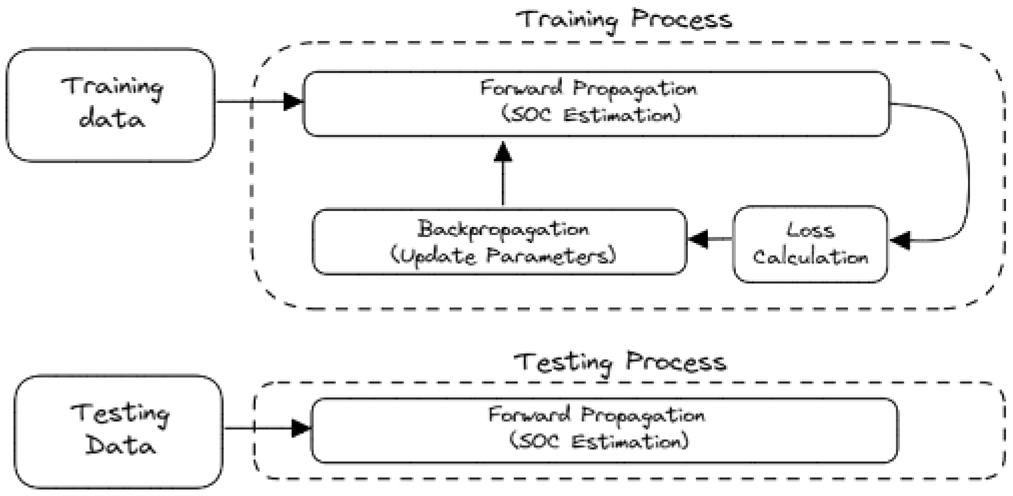
Training Process
Training data
Forward Propagation (SOC Estimation)
Backpropagation (Update Parameters)
Loss Calculation
Testing Process
Testing Data
Forward Propagation (SOC Estimation)

Fig. Neural network training and validation process overview

The training process of FNN consists of forward propagation, loss calculation and backpropagation. In forward propagation we just pass the inputs through the network and get predictions, then we calculate loss between the predictions and the ground truth and backpropagate the gradients calculated using the loss. And in the testing process we just forward propagate inputs to get predictions.

We used following six measurements as inputs to the network:

- Measured Voltage
- Measured Current
- Measured Temperature
- SWA of Voltage
- SWA of Current
- SWA of Temperature

SWA means sliding window average which was taken with a window size of 500sec. And also the SOC values are normalized before training so that values are between 0 and 1.

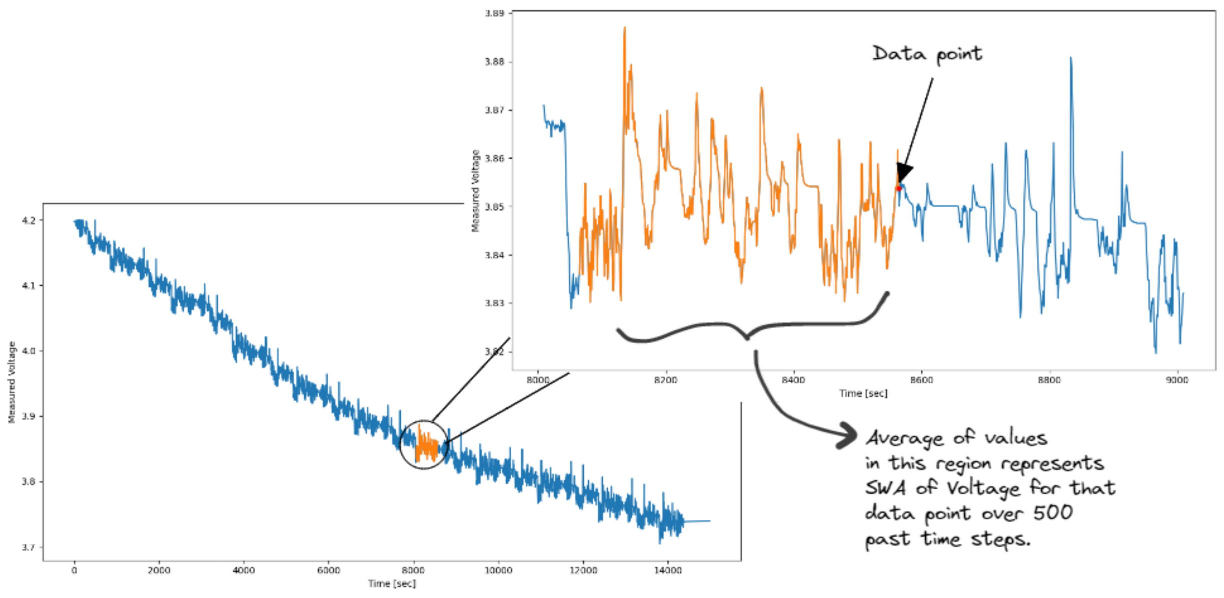

Fig. Example of a sliding window in measured voltage

3.3 Decision Tree Ensemble

Decision Tree is a type of Machine Learning algorithm which can be used both for classification and Regression. Decision Trees (DTs) are a non-parametric supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. A tree can be seen as a piecewise constant approximation.

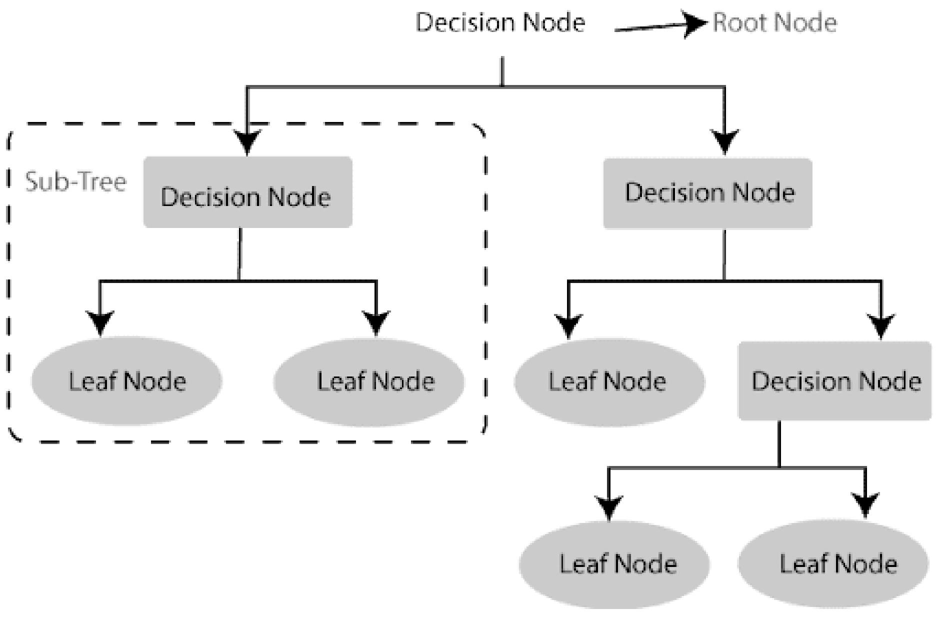

Fig. Block diagram of a decision tree

Decision trees often overfit the data. This is generally solved by using an ensemble of decision trees. Ensemble methods combine several decision trees to produce better predictive performance than utilizing a single decision tree. The main principle behind the ensemble model is that a group of weak learners come together to form a strong learner. There are two types of ensemble learning methods:

1) Bagging methods like Random Forest
2) Boosting methods like XGBoost

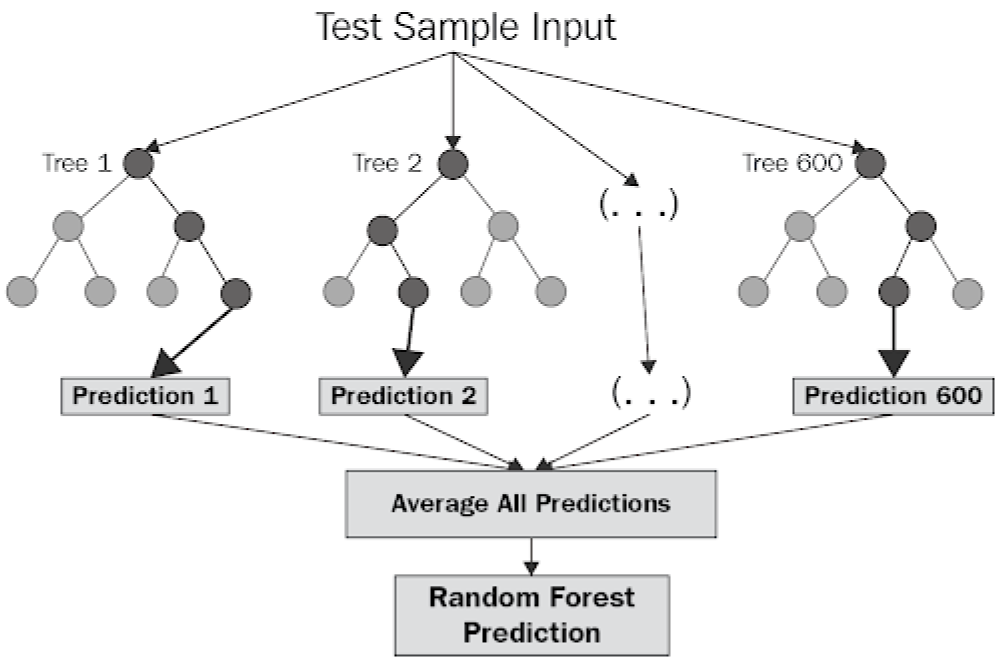

Fig. Decision tree ensemble

We used following six measurements as inputs to the ensemble model:

- Measured Voltage
- Measured Current
- Measured Temperature
- SWA of Voltage
- SWA of Current
- SWA of Temperature

SWA means sliding window average which was taken with a window size of 500sec. And also the SOC values are normalized before training so that values are between 0 and 1.

4. Experiments and results

Extended Kalman filter parameters were carefully tuned by trial and error. And when it was run on one of the drive cycle experiments LA92, estimated mean percentage error is 3.03%.

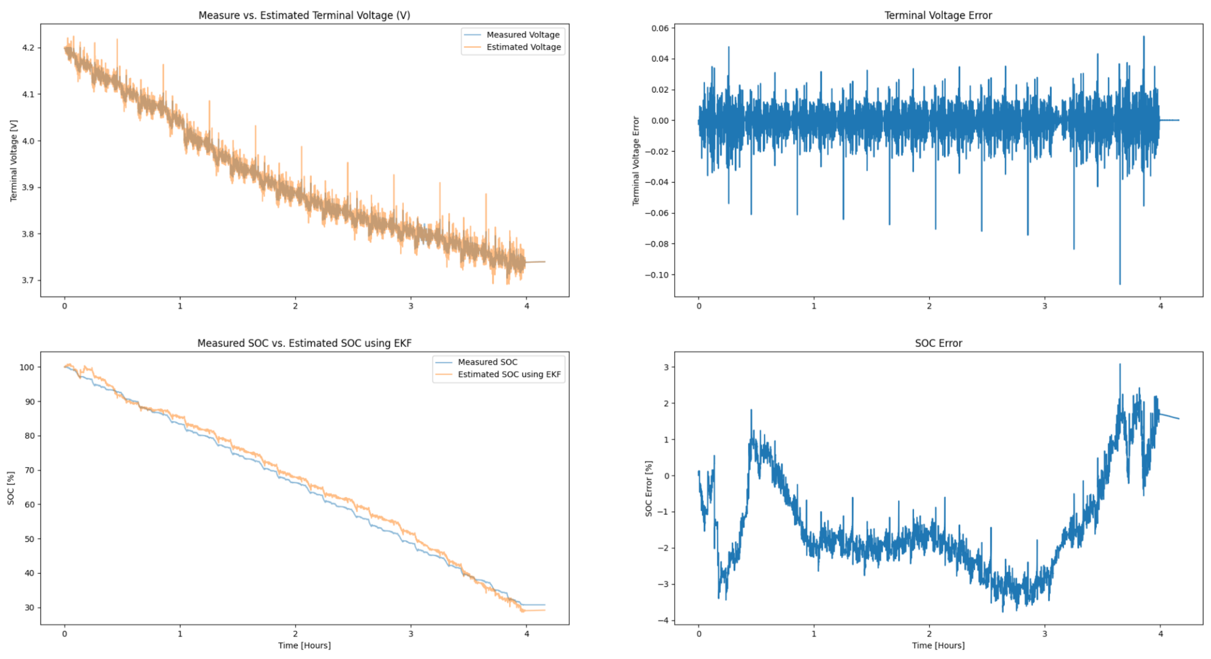

Fig. EKF predictions

A feedforward neural network was trained using following parameters: 6 inputs neurons, 3 hidden layers with (64, 128, 64) neurons respectively and with (Tanh, PRelu, PRelu) activations respectively, initial learning rate is 0.01, learning rate drop factor is 0.96, optimizer used to train is Adam, batch size is set to 1024, number of epochs is 100 and loss function used is Mean Squared Error.

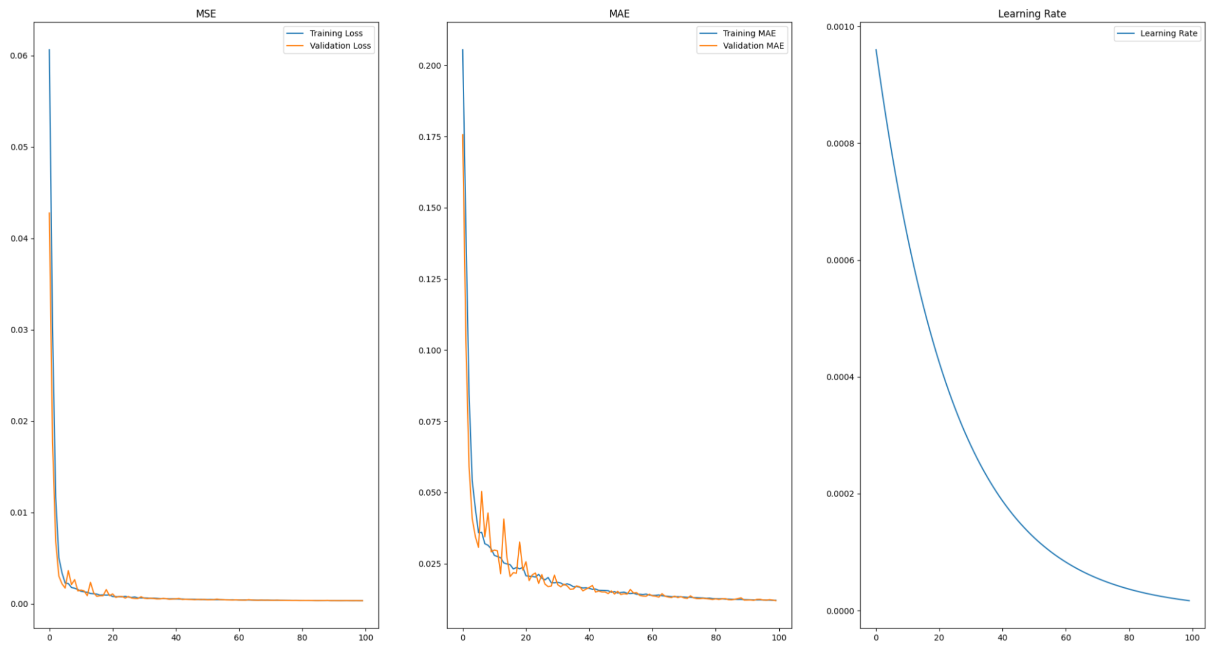

Fig. Training metrics and validation metrics

The trained feedforward neural network was run on one of the drive cycle experiments LA92, estimated mean percentage error is 1.20%.

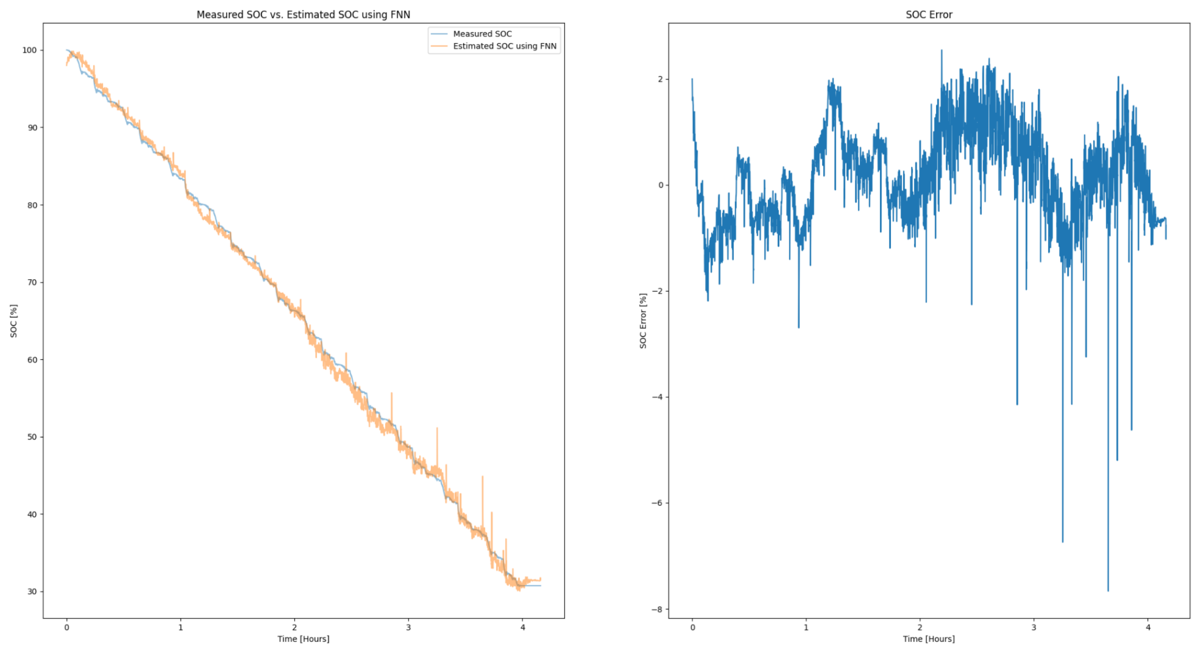

Fig. FNN predictions

XGBoost Regressor was trained with 1000 estimators and with a max depth of 6, on the processed data. Later the trained XGBoost Regressor was run on one of the drive cycle experiments LA92, estimated mean percentage error is 0.75%.

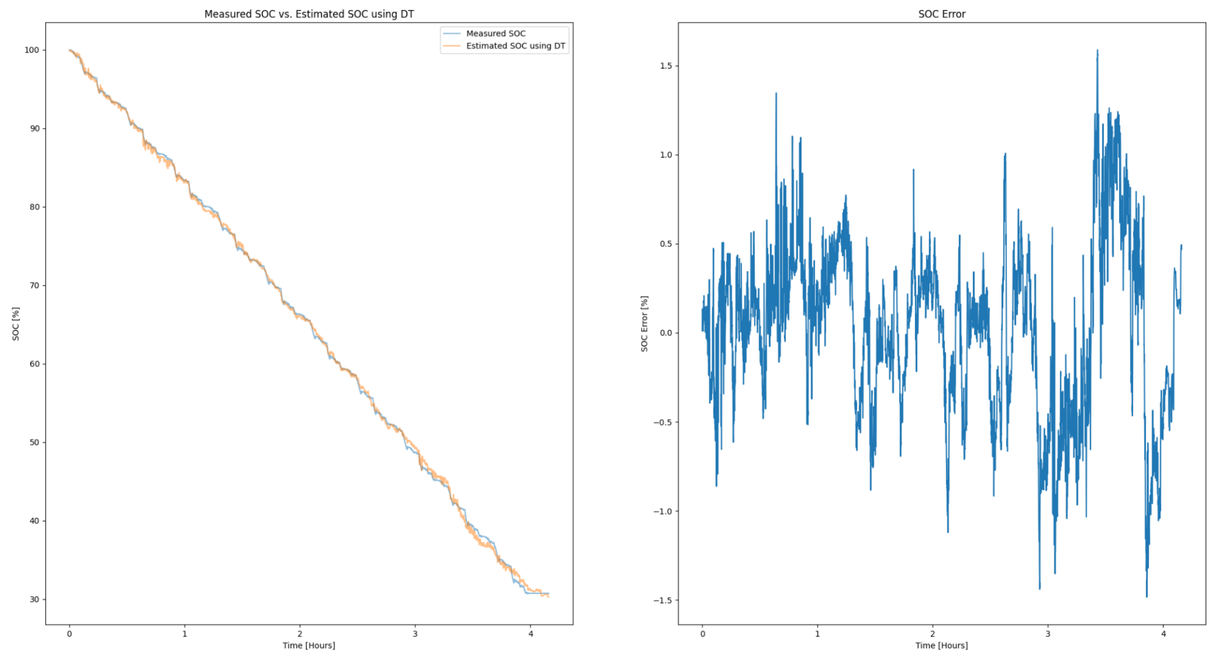

Fig. DT predictions

5. Conclusions

The percentage errors obtained in each of the methods is:

1) Error obtained in EKF: 3.03%
2) Error obtained in FNN: 1.20%
3) Error obtained in DTE: 0.75%

We observe that the Decision Tree Ensemble method works better than the other two methods in predicting the State of Charge.

All the code files are available at: https://github.com/vstark21/SOC_estimation

6. References

- How to Estimate Battery State of Charge using Deep Learning
- Overview of batteries State of Charge estimation methods
- https://data.mendeley.com/datasets/4fx8cjprxm/
- State of Charge Estimation Using Extended Kalman Filters for Battery Management System
- State of Charge Estimation based on Kalman Filter
- Battery State of Charge Estimation Using an Artificial Neural network

Appendix

EKF/extended_kalman_filter.py

```
import numpy as np
import pandas as pd
from scipy.interpolate import interp2d
from tqdm import tqdm

def run_ekf(
    data: pd.DataFrame,
    bat_model: pd.DataFrame,
    soc_ocv: pd.DataFrame,
):
    Current = data['Measured_Current_R'].values
    Vt_Actual = data['Measured_Voltage'].values
    Temperature = data['Measured_Temperature'].values

    SOC_Init = 1 # Initial SOC
    X = np.array([[SOC_Init], [0], [0]]) # State space x parameter
initialisation
    DeltaT = 1 # Sample time in seconds
    Qn_rated = 5. * 3600 # Ah to Amp-sec

    F_R0 = interp2d(bat_model['T'].values, bat_model.SOC.values,
bat_model.R0.values)
    F_R1 = interp2d(bat_model['T'].values, bat_model.SOC.values,
bat_model.R1.values)
    F_R2 = interp2d(bat_model['T'].values, bat_model.SOC.values,
bat_model.R2.values)
    F_C1 = interp2d(bat_model['T'].values, bat_model.SOC.values,
bat_model.C1.values)
    F_C2 = interp2d(bat_model['T'].values, bat_model.SOC.values,
bat_model.C2.values)

    SOCOCV = np.polyfit(soc_ocv['SOC'].values, soc_ocv['OCV'].values, 11)
    dSOCOCV = np.polyder(SOCOCV)

    n_x = X.shape[0]
    R_x = 2.5e-5
    P_x = np.array([
        [0.025, 0, 0],
        [0, 0.01, 0],
        [0, 0, 0.01]
    ])
    Q_x = np.array([
        [1.0e-6, 0, 0],
        [0, 1.0e-5, 0],
        [0, 0, 1.0e-5]
```

```
])

SOC_Estimated = []
Vt_Estimated = []
Vt_Error = []
ik = len(Current)

for k in tqdm(range(ik)):
    T = Temperature[k]
    U = Current[k]
    SOC = X[0, 0]
    V1 = X[1, 0]
    V2 = X[2, 0]

    # Evaluate the battery parameter
    # Functions for the current temperature and SOC
    R0 = F_R0(T, SOC).squeeze()
    R1 = F_R1(T, SOC).squeeze()
    R2 = F_R2(T, SOC).squeeze()
    C1 = F_C1(T, SOC).squeeze()
    C2 = F_C2(T, SOC).squeeze()

    OCV = np.polyval(SOCOCV, SOC)

    Tau_1 = C1 * R1
    Tau_2 = C2 * R2

    a1 = np.exp(-DeltaT / Tau_1)
    a2 = np.exp(-DeltaT / Tau_2)

    b1 = R1 * (1 - np.exp(-DeltaT / Tau_1))
    b2 = R2 * (1 - np.exp(-DeltaT / Tau_2))

    Terminal_Voltage = OCV - U * R0 - V1 - V2
    eta = 1

    dOCV = np.polyval(dSOCOCV, SOC)
    C_x = np.array([[dOCV, -1, -1]])

    Error_x = np.array([[Vt_Actual[k] - Terminal_Voltage]])

    Vt_Estimated.append(Terminal_Voltage)
    SOC_Estimated.append(X[0, 0])
    Vt_Error.append(Error_x.squeeze())

    A = np.array([
        [1, 0, 0],
        [0, a1, 0],
        [0, 0, a2]
    ])
    B = np.array([[-(eta * DeltaT / Qn_rated)], [b1], [b2]])
    X = (A @ X) + (B @ np.array([[U]]))
    P_x = (A @ P_x @ A.T) + Q_x

    a = (C_x @ P_x @ C_x.T) + R_x
    KalmanGain_x = P_x @ C_x.T @ np.linalg.inv(a)
    X = X + (KalmanGain_x @ Error_x)
    P_x = (np.eye(n_x, n_x) - (KalmanGain_x @ C_x)) @ P_x

SOC_Estimated = np.array(SOC_Estimated)
Vt_Estimated = np.array(Vt_Estimated)
Vt_Error = np.array(Vt_Error)

return SOC_Estimated, Vt_Estimated, Vt_Error
```

EKF/utils.py

```
import scipy.io
import numpy as np
import pandas as pd

def load_data(
    mat_file: str
) -> pd.DataFrame:
    data = scipy.io.loadmat(mat_file)
    meas = data['meas']
    dtypes = meas.dtype
    meas = np.squeeze(meas).tolist()
    data_dict = {}
    for i, name in enumerate(dtypes.names):
        data_dict[name] = np.array(meas[i]).squeeze()

    data_dict = pd.DataFrame(data_dict)
    data_dict.rename(columns={
        'Time': 'RecordingTime',
        'Voltage': 'Measured_Voltage',
        'Current': 'Measured_Current',
        'Battery_Temp_degC': 'Measured_Temperature',
    }, inplace=True)
    return data_dict
```

EKF/main.py

```
import numpy as np
from utils import *
import matplotlib.pyplot as plt
from extended_kalman_filter import run_ekf

DATA_FILE = '../04-20-19_05.34 780_LA92_25degC_Turnigy_Graphene.mat'
BATTERY_MODEL = '../data/BatteryModel.csv'
SOC_OCV = '../data/SOC-OCV.csv'

if __name__ == '__main__':

    LiPoly = load_data(DATA_FILE)

    # Battery capacity in Ah taken from data
    nominal_cap = 5.
    # Calculate the SOC using coloumb counting for comparision
    LiPoly['Measured_SOC'] = (nominal_cap + LiPoly['Ah']) * 100. / nominal_cap

    # Resample data
    LiPoly = LiPoly.loc[0::10]

    # Discharging: +ve current, charging: -ve current
    LiPoly['Measured_Current_R'] = LiPoly['Measured_Current'] * -1

    # Converting seconds to hours
    LiPoly['RecordingTime_Hours'] = LiPoly['RecordingTime'] / 3600

    # Load battery model and soc-ocv data
    battery_model = pd.read_csv(BATTERY_MODEL)
    soc_ocv = pd.read_csv(SOC_OCV)

    print(battery_model.head())
    print(soc_ocv.head())

    SOC_Estimated, Vt_Estimated, Vt_Error = run_ekf(
        LiPoly, battery_model, soc_ocv
    )

    percentage_error = np.abs((LiPoly['Measured_SOC'] - SOC_Estimated * 100) / 
LiPoly['Measured_SOC']) * 100
    print('Mean percentage error: {:.2f}%'.format(np.mean(percentage_error)))

    # Plot the results
    fig, axes = plt.subplots(2, 2)

    axes[0, 0].plot(LiPoly['RecordingTime_Hours'], LiPoly['Measured_Voltage'], 
label='Measured Voltage', alpha=0.5)
```

```
    axes[0, 0].plot(LiPoly['RecordingTime_Hours'], Vt_Estimated,
label='Estimated Voltage', alpha=0.5)
    axes[0, 0].set_ylabel('Terminal Voltage [V]')
    axes[0, 0].set_title('Measure vs. Estimated Terminal Voltage (V)')
    axes[0, 0].legend()

    axes[0, 1].plot(LiPoly['RecordingTime_Hours'], Vt_Error)
    axes[0, 1].set_ylabel('Terminal Voltage Error')
    axes[0, 1].set_title('Terminal Voltage Error')

    axes[1, 0].plot(LiPoly['RecordingTime_Hours'], LiPoly['Measured_SOC'],
label='Measured SOC', alpha=0.5)
    axes[1, 0].plot(LiPoly['RecordingTime_Hours'], SOC_Estimated * 100.,
label='Estimated SOC using EKF', alpha=0.5)
    axes[1, 0].set_ylabel('SOC [%]')
    axes[1, 0].set_xlabel('Time [Hours]')
    axes[1, 0].set_title('Measured SOC vs. Estimated SOC using EKF')
    axes[1, 0].legend()

    # Plot soc error
    axes[1, 1].plot(LiPoly['RecordingTime_Hours'], (LiPoly['Measured_SOC'] -
SOC_Estimated * 100.))
    axes[1, 1].set_ylabel('SOC Error [%]')
    axes[1, 1].set_xlabel('Time [Hours]')
    axes[1, 1].set_title('SOC Error')

    plt.show()
```

FNN/create_data.py

```
from pathlib import Path
import numpy as np
import pandas as pd
from utils import load_data
from tqdm import tqdm

TEMPERATURES = [-10, -20, 0, 10, 25, 40]
DRIVE_CYCLES = ['UDDS', 'HWFET', 'LA92', 'US06']
WINDOW_SIZE = 500

def preprocess_data(
    df: pd.DataFrame
):
    # Sort data by time
    df = df.sort_values(by='RecordingTime').reset_index(drop=True)

    # Resample data at 1Hz
    df = df.loc[::10]
    # Apply rolling window to data on voltage, current and temperature
    df['Avg_Measured_Voltage'] = df['Measured_Voltage'].rolling(WINDOW_SIZE,
min_periods=1).mean()
    df['Avg_Measured_Current'] = df['Measured_Current'].rolling(WINDOW_SIZE,
min_periods=1).mean()
    df['Avg_Measured_Temperature'] =
df['Measured_Temperature'].rolling(WINDOW_SIZE, min_periods=1).mean()

    return df

def create_data(
    data_dir
) -> pd.DataFrame:

    data_dir = Path(data_dir)
    total_data = []

    for t in tqdm(TEMPERATURES):
        t_dir = data_dir / f"{t} degC"
        assert t_dir.exists(), f"{t_dir} does not exist"

        for d in DRIVE_CYCLES:
            for f in t_dir.glob(f"*{d}*.mat"):

                cur_data = load_data(f)
                cur_data = preprocess_data(cur_data)
                total_data.append(cur_data)

    total_data = pd.concat(total_data)
    total_data.reset_index(drop=True, inplace=True)

    return total_data


if __name__ == '__main__':
    data = create_data('../data/Turnigy Graphene')
    data.to_csv('turnigy_graphene_data.csv', index=False)
```

FNN/main.py

```
import numpy as np
from utils import *
import matplotlib.pyplot as plt
from create_data import preprocess_data
from train import FEATURES
import tensorflow as tf
from tensorflow import keras
```

```
DATA_FILE = '../04-20-19_05.34 780_LA92_25degC_Turnigy_Graphene.mat'

if __name__ == '__main__':

    LiPoly = load_data(DATA_FILE)
    LiPoly = preprocess_data(LiPoly)

    # Battery capacity in Ah taken from data
    nominal_cap = 5.

    # Calculate the SOC using coloumb counting for comparision
    LiPoly['Measured_SOC'] = (nominal_cap + LiPoly['Ah']) * 100. / nominal_cap

    # Converting seconds to hours
    LiPoly['RecordingTime_Hours'] = LiPoly['RecordingTime'] / 3600

    model = keras.models.load_model('best_turnigy_graphene_model.h5')
    SOC_Estimated = model.predict(LiPoly[FEATURES].values) * 100.
    SOC_Estimated = np.squeeze(SOC_Estimated)
    SOC_Estimated = simple_exponential_smoothing(SOC_Estimated, 0.25)

    percentage_error = np.abs(LiPoly['Measured_SOC'] - SOC_Estimated) /
LiPoly['Measured_SOC'] * 100.
    print('Mean percentage error: {:.2f}%'.format(np.mean(percentage_error)))

    # Plot the results
    fig, axes = plt.subplots(1, 2)

    axes[0].plot(LiPoly['RecordingTime_Hours'], LiPoly['Measured_SOC'],
label='Measured SOC', alpha=0.5)
    axes[0].plot(LiPoly['RecordingTime_Hours'], SOC_Estimated,
label='Estimated SOC using FNN', alpha=0.5)
    axes[0].set_ylabel('SOC [%]')
    axes[0].set_xlabel('Time [Hours]')
    axes[0].set_title('Measured SOC vs. Estimated SOC using FNN')
    axes[0].legend()

    # Plot soc error
    axes[1].plot(LiPoly['RecordingTime_Hours'], (LiPoly['Measured_SOC'] -
SOC_Estimated))
    axes[1].set_ylabel('SOC Error [%]')
    axes[1].set_xlabel('Time [Hours]')
    axes[1].set_title('SOC Error')

    plt.show()
```

FNN/model.py

```
import tensorflow as tf
from tensorflow import keras
from keras import layers

def create_model(input_shape):
    model = keras.Sequential([
        layers.Input(shape=input_shape),
        layers.Dense(64, activation='tanh'),
        layers.Dense(128),
        layers.PReLU(),
        layers.Dense(64),
        layers.PReLU(),
        layers.Dense(1),
        layers.ReLU(max_value=1)
    ])

    model.compile(
        optimizer=keras.optimizers.Adam(lr=0.001),
        loss='mse',
        metrics=['mae']
    )

    return model

def scheduler(epoch, lr):
    gamma = 0.96
    lr *= gamma
    lr = max(lr, 1e-5)
    return lr

def train_model(model, x_train, x_val, y_train, y_val, epochs, batch_size):
    history = model.fit(
        x=x_train,
        y=y_train,
        batch_size=batch_size,
        epochs=epochs,
        validation_data=(x_val, y_val),
        callbacks=[keras.callbacks.LearningRateScheduler(scheduler)]
    )
    return history, model
```

FNN/train.py

```
import numpy as np
import pandas as pd
from pathlib import Path
from create_data import create_data
```

```python
from model import create_model, train_model
from sklearn.model_selection import train_test_split

import tensorflow as tf
import matplotlib.pyplot as plt

NOMINAL_CAP = 5.0
DATA_FILE = Path('turnigy_graphene_data.csv')
FEATURES = ['Measured_Voltage', 'Measured_Current',
'Measured_Temperature',  'Avg_Measured_Voltage', 'Avg_Measured_Current',
'Avg_Measured_Temperature']
LABEL = 'Measured_SOC'
EPOCHS = 100
BATCH_SIZE = 1024

if __name__ == '__main__':
    if not DATA_FILE.exists():
        data = create_data('../data/Turnigy Graphene')
    else:
        data = pd.read_csv(DATA_FILE)

    data['Measured_SOC'] = (NOMINAL_CAP + data['Ah']) * 100. / NOMINAL_CAP
    data['RecordingTime_Hours'] = data['RecordingTime'] / 3600

    print(f"Total data shape: {data.shape}")

    train, val = train_test_split(data, test_size=0.25, random_state=42)

    print(f"Train data shape: {train.shape}")
    print(f"Test data shape: {val.shape}")

    x_train = train[FEATURES].values
    y_train = train[LABEL].values / 100.
    x_val = val[FEATURES].values
    y_val = val[LABEL].values / 100.

    model = create_model((len(FEATURES)))
    history, model = train_model(
        model,
        x_train,
        x_val,
        y_train,
        y_val,
        epochs=EPOCHS,
        batch_size=BATCH_SIZE
    )
    model.save('turnigy_graphene_model.h5')
    fig, axes = plt.subplots(1, 3)
    axes[0].plot(history.history['loss'], label='Training Loss')
    axes[0].plot(history.history['val_loss'], label='Validation Loss')
    axes[0].set_title('MSE')
    axes[0].legend()

    axes[1].plot(history.history['mae'], label='Training MAE')
    axes[1].plot(history.history['val_mae'], label='Validation MAE')
    axes[1].set_title('MAE')
    axes[1].legend()

    axes[2].plot(history.history['lr'], label='Learning Rate')
    axes[2].set_title('Learning Rate')
    axes[2].legend()

    plt.show()
```

FNN/utils.py

```python
import scipy.io
import numpy as np
import pandas as pd

def load_data(
    mat_file: str
) -> pd.DataFrame:
    data = scipy.io.loadmat(mat_file)
    meas = data['meas']
    dtypes = meas.dtype
    meas = np.squeeze(meas).tolist()
    data_dict = {}
    for i, name in enumerate(dtypes.names):
        data_dict[name] = np.array(meas[i]).squeeze()

    data_dict = pd.DataFrame(data_dict)
    data_dict.rename(columns={
        'Time': 'RecordingTime',
        'Voltage': 'Measured_Voltage',
        'Current': 'Measured_Current',
        'Battery_Temp_degC': 'Measured_Temperature',
    }, inplace=True)
    return data_dict

def simple_exponential_smoothing(arr, alpha=0.9):
    """
    Simple exponential smoothing
    """
    smoothed = np.zeros(len(arr))
    smoothed[0] = arr[0]
    for i in range(1, len(arr)):
        smoothed[i] = alpha * arr[i] + (1 - alpha) * smoothed[i - 1]
    return smoothed
```

DT/create_data.py

```
from pathlib import Path
import numpy as np
import pandas as pd
from utils import load_data
from tqdm import tqdm

TEMPERATURES = [-10, -20, 0, 10, 25, 40]
DRIVE_CYCLES = ['UDDS', 'HWFET', 'LA92', 'US06']
WINDOW_SIZE = 500

def preprocess_data(
    df: pd.DataFrame
):
    # Sort data by time
    df = df.sort_values(by='RecordingTime').reset_index(drop=True)

    # Resample data at 1Hz
    df = df.loc[::10]

    # Apply rolling window to data on voltage, current and temperature
    df['Avg_Measured_Voltage'] = df['Measured_Voltage'].rolling(WINDOW_SIZE,
min_periods=1).mean()
    df['Avg_Measured_Current'] = df['Measured_Current'].rolling(WINDOW_SIZE,
min_periods=1).mean()
    df['Avg_Measured_Temperature'] =
df['Measured_Temperature'].rolling(WINDOW_SIZE, min_periods=1).mean()

    return df

def create_data(
    data_dir
) -> pd.DataFrame:

    data_dir = Path(data_dir)
    total_data = []

    for t in tqdm(TEMPERATURES):
        t_dir = data_dir / f"{t} degC"
        assert t_dir.exists(), f"{t_dir} does not exist"
        for d in DRIVE_CYCLES:
            for f in t_dir.glob(f"*{d}*.mat"):

                cur_data = load_data(f)
                cur_data = preprocess_data(cur_data)
                total_data.append(cur_data)

    total_data = pd.concat(total_data)
    total_data.reset_index(drop=True, inplace=True)

    return total_data


if __name__ == '__main__':
    data = create_data('../data/Turnigy Graphene')
    data.to_csv('turnigy_graphene_data.csv', index=False)
```

DT/main.py

```
import pickle
import numpy as np
from utils import *
import matplotlib.pyplot as plt
from create_data import preprocess_data
from train import FEATURES, SAVE_FILE

DATA_FILE = '../04-20-19_05.34 780_LA92_25degC_Turnigy_Graphene.mat'

if __name__ == '__main__':

    LiPoly = load_data(DATA_FILE)
    LiPoly = preprocess_data(LiPoly)

    # Battery capacity in Ah taken from data
    nominal_cap = 5.

    # Calculate the SOC using coloumb counting for comparision
    LiPoly['Measured_SOC'] = (nominal_cap + LiPoly['Ah']) * 100. / nominal_cap

    # Converting seconds to hours
    LiPoly['RecordingTime_Hours'] = LiPoly['RecordingTime'] / 3600

    model = pickle.load(open(SAVE_FILE, "rb"))
    SOC_Estimated = model.predict(LiPoly[FEATURES].values) * 100.
    SOC_Estimated = np.squeeze(SOC_Estimated)
    SOC_Estimated = simple_exponential_smoothing(SOC_Estimated, 0.25)
    percentage_error = np.abs(LiPoly['Measured_SOC'] - SOC_Estimated) /
LiPoly['Measured_SOC'] * 100.
    print('Mean percentage error: {:.2f}%'.format(np.mean(percentage_error)))

    # Plot the results
    fig, axes = plt.subplots(1, 2)

    axes[0].plot(LiPoly['RecordingTime_Hours'], LiPoly['Measured_SOC'],
label='Measured SOC', alpha=0.5)
    axes[0].plot(LiPoly['RecordingTime_Hours'], SOC_Estimated,
label='Estimated SOC using DT', alpha=0.5)
    axes[0].set_ylabel('SOC [%]')
    axes[0].set_xlabel('Time [Hours]')
    axes[0].set_title('Measured SOC vs. Estimated SOC using DT')
    axes[0].legend()

    # Plot soc error
    axes[1].plot(LiPoly['RecordingTime_Hours'], (LiPoly['Measured_SOC'] -
SOC_Estimated))
    axes[1].set_ylabel('SOC Error [%]')
    axes[1].set_xlabel('Time [Hours]')
    axes[1].set_title('SOC Error')

    plt.show()
```

DT/train.py

```
import pickle
import numpy as np
import pandas as pd
from pathlib import Path
from xgboost import XGBRegressor
from create_data import create_data
from utils import compute_metrics
from sklearn.model_selection import train_test_split

NOMINAL_CAP = 5.0
DATA_FILE = Path('turnigy_graphene_data.csv')
FEATURES = ['Measured_Voltage', 'Measured_Current',
'Measured_Temperature',  'Avg_Measured_Voltage', 'Avg_Measured_Current',
'Avg_Measured_Temperature']
LABEL = 'Measured_SOC'
SAVE_FILE = Path('best_model.pkl')

if __name__ == '__main__':
    if not DATA_FILE.exists():
        data = create_data('../data/Turnigy Graphene')
        data.to_csv(DATA_FILE, index=False)
```

```python
    else:
        data = pd.read_csv(DATA_FILE)

    data[LABEL] = (NOMINAL_CAP + data['Ah']) * 100. / NOMINAL_CAP

    train, val = train_test_split(data, test_size=0.25, random_state=42)

    x_train = train[FEATURES].values
    y_train = train[LABEL].values / 100.
    x_val = val[FEATURES].values
    y_val = val[LABEL].values / 100.

    model = XGBRegressor(n_estimators=1000)
    model.fit(x_train, y_train)

    train_metrics = compute_metrics(model.predict(x_train), y_train)
    val_metrics = compute_metrics(model.predict(x_val), y_val)

    print(f"TRAIN METRICS: {train_metrics}")
    print(f"VAL METRICS: {val_metrics}")

    pickle.dump(model, open(SAVE_FILE, "wb"))
```

DT/utils.py

```python
import scipy.io
import numpy as np
import pandas as pd
from sklearn.metrics import mean_squared_error, mean_absolute_error

def load_data(
    mat_file: str
) -> pd.DataFrame:
    data = scipy.io.loadmat(mat_file)
    meas = data['meas']
    dtypes = meas.dtype
    meas = np.squeeze(meas).tolist()
    data_dict = {}
    for i, name in enumerate(dtypes.names):
        data_dict[name] = np.array(meas[i]).squeeze()

    data_dict = pd.DataFrame(data_dict)
    data_dict.rename(columns={
        'Time': 'RecordingTime',
        'Voltage': 'Measured_Voltage',
        'Current': 'Measured_Current',
        'Battery_Temp_degC': 'Measured_Temperature',
    }, inplace=True)
    return data_dict

def compute_metrics(preds, labels):
    return {
        'mse': mean_squared_error(preds, labels),
        'mae': mean_absolute_error(preds, labels)
    }

def simple_exponential_smoothing(arr, alpha=0.9):
    """
    Simple exponential smoothing
    """
    smoothed = np.zeros(len(arr))
    smoothed[0] = arr[0]
    for i in range(1, len(arr)):
        smoothed[i] = alpha * arr[i] + (1 - alpha) * smoothed[i - 1]
    return smoothed
```