Machine Learning Methods for Early Prediction of End-of-Life for Lithium-ion Batteries

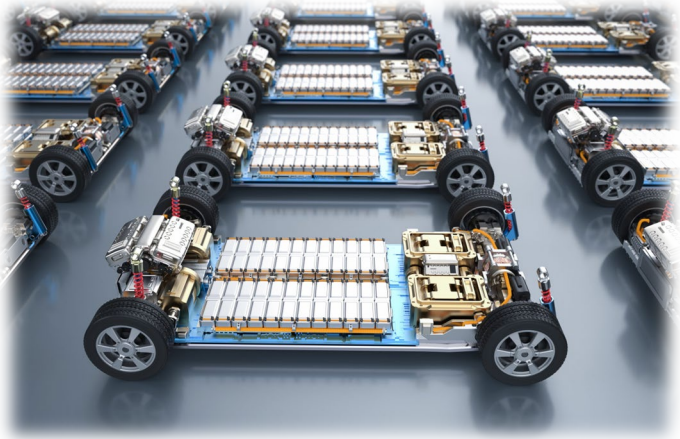
James Chapman
CIS 732 Machine Learning – Term Project
Kansas State University
jachapman@ksu.edu

Overview

- Lithium-Ion Batteries Introduction
- The Battery Data Set
  - Severson et al. 2019 (MIT)
  - 2 types of features (1D & 2D)
- Machine Learning Models
  - Support Vector Regression (SVR)
  - Multilayer Perceptron (MLP)
  - Long-Short-Term-Memory RNN (LSTM)
  - Convolutional Neural Network (CNN)
- Results & Conclusion

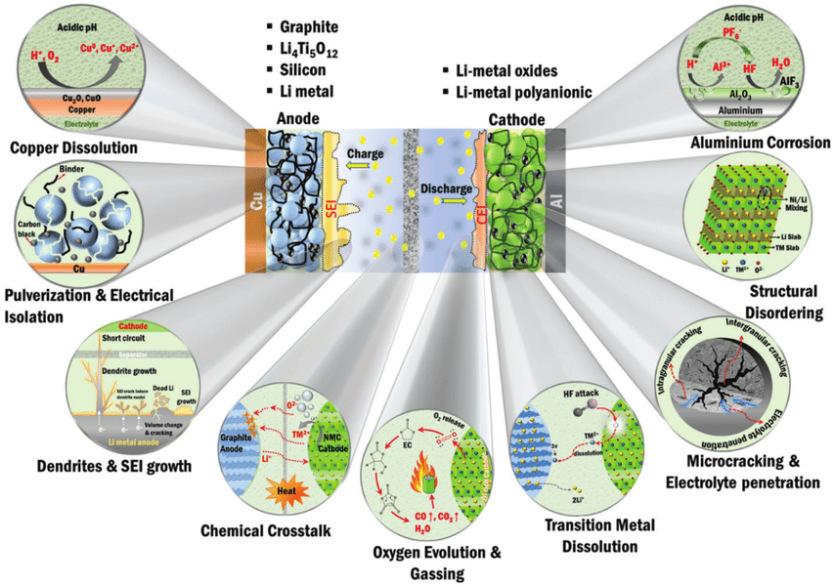
Lithium-Ion Batteries

- Big future
  - Environmental concerns / Renewability
  - Electric vehicles (EVs)
  - Consumer electronics
  - Grid-scale support



Battery Degradation

- Complex electrochemical mechanisms
- Highly irregular behavior
- End-of-life
  - Wide range/high variability
  - Hard to predict
  - Severe consequences



Overview of the major degradation mechanisms

Narayan, R., Laberty-Robert, C., Pelta, J., Tarascon, J. M., & Dominko, R. (2021). Self-Healing: An emerging technology for next-generation smart batteries. *Advanced Energy Materials*, *12*(17), 2102652. https://doi.org/10.1002/aenm.202102652

KANSAS STATE UNIVERSITY

Data-driven modeling

- Mechanism-Agnostic
  - Black box
- Machine learning & artificial intelligence
  - Captures behaviors and patterns
  - Harness information

<u>Data sparsity</u>

  - Expensive
  - Experiments take months

The Battery Data Set

- Severson et al. 2019 (MIT)
  - Battery Model - APR18650M1A
- Largest open-source data set of its kind
- “Cycles”
  - Charged/Discharged until end-of-life
  - Variety of conditions mimic real-world

Severson, K. A., Attia, P. M., Jin, N., Perkins, N., Jiang, B., Yang, Z., Chen, M. H., Aykol, M., Herring, P. K., Fraggedakis, D., Bazant, M. Z., Harris, S. J., Chueh, W. C., & Braatz, R. D. (2019). Data-driven prediction of battery cycle life before capacity degradation. *Nature Energy*, *4*(5), 383–391. doi.org/10.1038/s41560-019-0356-8

Severson et al. 2019

- End-of-life
  - No longer capable of 80% capacity
  - Integer (~300-2500)
  - Number of “cycles”
  - Target of machine learning (y)

GOAL

- Predict end-of-life
- Data from first 100 cycles (or less)

Tabular Data

- Used in original paper
- Calculated summary statistics

  "1-dimensional"
- "FULL" feature set
  - 10 features
- "DISCHARGE" feature set
  - 6 features
  - Using only discharge capacity-voltage data

TABLE I
SUMMARY FEATURES FROM DISCHARGE VOLTAGE CURVES USING THE 10TH AND 100TH CYCLES (ΔQ100-10(V))

- Minimum ($\Delta Q_{100\text{-}10}(V)$)
- Variance ($\Delta Q_{100\text{-}10}(V)$)
- Skewness ($\Delta Q_{100\text{-}10}(V)$)
- Kurtosis ($\Delta Q_{100\text{-}10}(V)$)
- Discharge capacity of cycle 2
- Max discharge capacity - Discharge capacity of cycle 2

Tabular Data

In [135…

```
dataset
```

Out[135…

	cell	cell_life	minimum_dQ_100_10	variance_dQ_100_10	skewness_dQ_100_10	kurtosis_dQ_100_10	mean_dQ_100_10	Slope_Cap_Fade_2_100	Intercept_Cap_Fade
0	b1c0	1852.0	-2.072648	-5.014861	-0.274041	0.129790	-2.541602	1.692000e-05	1.
1	b1c1	2160.0	-1.958457	-5.013960	-0.367163	0.012464	-2.387257	5.535293e-06	1.
2	b1c2	2237.0	-1.764058	-4.737000	0.033502	-0.457627	-2.348070	1.029563e-05	1.
3	b1c3	1434.0	-1.722149	-4.442613	-0.357486	0.039579	-2.127507	1.713635e-05	1.
4	b1c4	1709.0	-1.855177	-4.647744	-0.440634	0.125101	-2.240332	1.899432e-05	1.
...	...	...	...	...	...	...	...	...	
119	b3c39	1156.0	-1.758008	-4.454635	-0.533817	0.001307	-2.191895	-6.583238e-06	1.
120	b3c40	796.0	-1.656517	-4.295108	-0.501096	0.056338	-2.041969	-7.692147e-06	1.
121	b3c41	786.0	-1.616310	-4.219509	-0.447169	0.026586	-2.011082	9.732245e-06	1.
122	b3c44	940.0	-1.585275	-4.131496	-0.611514	0.092860	-1.949579	4.689737e-06	1.
123	b3c45	1801.0	-1.783097	-4.520851	-0.483937	0.070824	-2.146804	-1.258454e-07	1.

124 rows × 18 columns

KANSAS STATE UNIVERSITY

Discharge Capacity-Voltage Data

- Capacity values (Ah) during “discharge”
- Interpolated at 100 evenly spaced voltages (3.5V-2V)

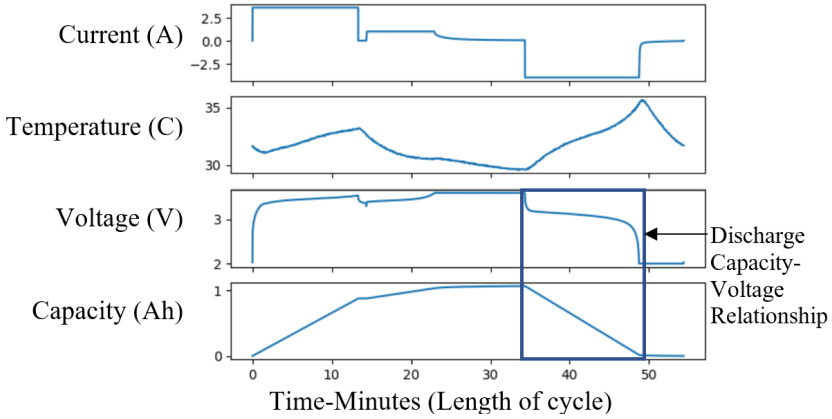


Fig. 2. Timeseries Data ,
Sample - cycle 1 of battery ‘b1c0’

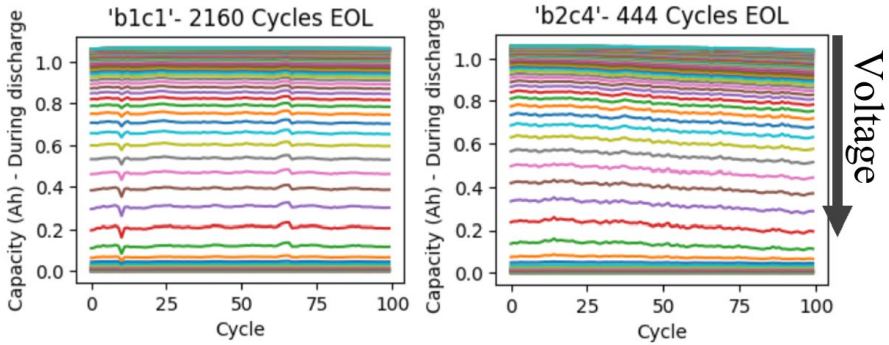


Fig. 3. Discharge Capacity-Voltage Curves
Comparison of batteries with high and low end-of-life

KANSAS STATE UNIVERSITY

Discharge Capacity-Voltage Data

- Two-dimensional
  - 99 cycles X 100 voltages
    - (skip cycle 1)
  - 9900 capacities (Ah) per battery
- Used in follow-up paper
  - Attia et al.
  - (and other papers)

	Cycle 2	Cycle 3	…	Cycle 100
3.5 V	Ah	Ah	Ah	Ah
3.485 V	Ah	Ah	Ah	Ah
…	Ah	Ah	Ah	Ah
2 V	Ah	Ah	Ah	Ah

100 evenly spaced voltages

Machine Learning Models

- Tabular data (1D)
  - Support Vector Regression (SVR)
  - Multilayer Perceptron (MLP)
- Discharge capacity-voltage data (2D)
  - Long-Short-Term-Memory RNN (LSTM)
  - Convolutional Neural Network (CNN)

Support Vector Regression (SVR)

- SVM- Projection to higher dimensional feature space
  - Kernel (Sigmoid, RBF, etc.)
  - Hyperplane ▸ real value (Regression)
- Hyperparameters (cross-validation)
  - Epsilon (margin)
  - C (L2 regularization)
  - Gamma

RBF Kernel: $K(x, y) = e^{-}(\gamma||x - y||^2)$

Sigmoid Kernel: $K(x, y) = tanh(\gamma.x^T y + r)$

SVR

- Code

sklearn.svm.SVR

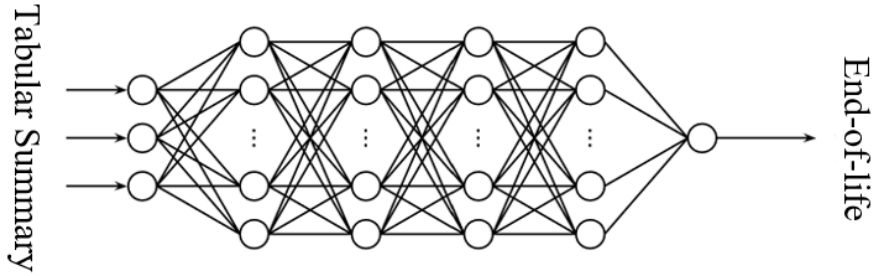
```
def SVRmodel(features,target,params):
    X_train=Training_Data[features].to_numpy()
    y_train=Training_Data[target].to_numpy()
    X_test=Testing_Data[features].to_numpy()
    y_test=Testing_Data[target].to_numpy()
    X_sec=Secondary_Data[features].to_numpy()
    y_sec=Secondary_Data[target].to_numpy()

    scaler = preprocessing.StandardScaler().fit(X_train)
    X_train = scaler.transform(X_train)
    X_test = scaler.transform(X_test)
    X_sec = scaler.transform(X_sec)

##################

    model = SVR()
    grid=GridSearchCV(estimator=model,param_grid=params,cv=40)

#################

    grid.fit(X_train, y_train)
    print(grid.best_params_)

    y_pred_train=10**(grid.predict(X_train))
    y_pred_test=10**(grid.predict(X_test))
    y_pred_sec=10**(grid.predict(X_sec))
```

SVR Hyperparameters

```
SVR_dismod_hparams={'epsilon':[.01,0.1,1],
            'gamma':[.009,.001, 0.01, 0.1, 1, 10],
            'kernel':["sigmoid","linear","rbf"]}

SVR_full_hparams={'epsilon':[.01,0.1,1],
            'gamma':['scale',.009,.001, 0.01, 0.1, 1, 10],
            'kernel':["linear","rbf"]} # Sigmoid & Scale cause problems
```

Multilayer Perceptron (MLP)

- Neural network for tabular data
- Nonlinear activation
  - Relu
- Optimization
  - Adaptive momentum
- Regularization
  - L2 (alpha hyperparameter)



MLP

- Code

sklearn.neural_network

MLPRegressor

MLP Hyperparameters

The hidden layer Sizes used for cross validation were chosen to match the Murphy's PML1 page 429, Neural Networks For Tabular data, Wh
website by Brendan Hasz. As well as the shape of the MLP used for MNIST(Page 427). The size to was reduced match this data's Small featu

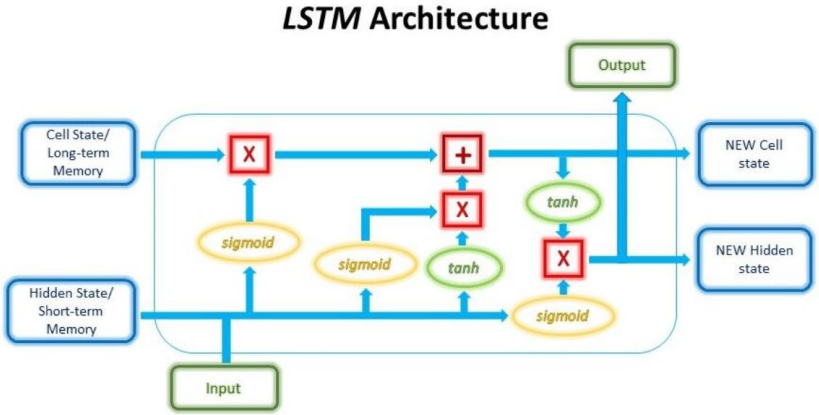
https://brendanhasz.github.io/2019/07/23/bayesian-density-net.html

[6]:
```
MLP_dismod_hparams={'hidden_layer_sizes':[[512,128,64,32],[784,128,128,10],[128,32,16],[256,128,64,32,16],
                                          [16, 32, 64, 128, 64, 32, 16],[16,32,64,32,16],[256, 64, 32, 16],[32,64,32,16]],
                    'alpha':[.001, 0.01, 0.1, 1, 10]}

MLP_full_hparams={'hidden_layer_sizes':[[512,128,64,32],[784,128,128,10],[128,32,16],[256,128,64,32,16],
                                        [16, 32, 64, 128, 64, 32, 16],[16,32,64,32,16],[256, 64, 32, 16],[32,64,32,16]],
                  'alpha':[.001, 0.01, 0.1, 1, 10]}
```

```
def MLPmodel(features,target,params):
    X_train=Training_Data[features].to_numpy()
    y_train=Training_Data[target].to_numpy()
    X_test=Testing_Data[features].to_numpy()
    y_test=Testing_Data[target].to_numpy()
    X_sec=Secondary_Data[features].to_numpy()
    y_sec=Secondary_Data[target].to_numpy()

    scaler = preprocessing.StandardScaler().fit(X_train)
    X_train = scaler.transform(X_train)
    X_test = scaler.transform(X_test)
    X_sec = scaler.transform(X_sec)

##################

    model = MLPRegressor(random_state=0, max_iter=10000)
    grid = GridSearchCV(estimator=model, param_grid=params)

##################

    grid.fit(X_train, y_train)
    print(grid.best_params_)

    y_pred_train=10**(grid.predict(X_train))
    y_pred_test=10**(grid.predict(X_test))
    y_pred_sec=10**(grid.predict(X_sec))
```

Long-Short-Term-Memory RNN (LSTM)

- Timeseries data
- Cell state
- Hidden state



Inner Workings of the LSTM cell

$$
\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}
$$

Fig. 6. The sequence of calculations at each cell in LSTM [10] (input gate, forget gate, cell gait, & output gate)

Long-Short-Term-Memory RNN (LSTM)

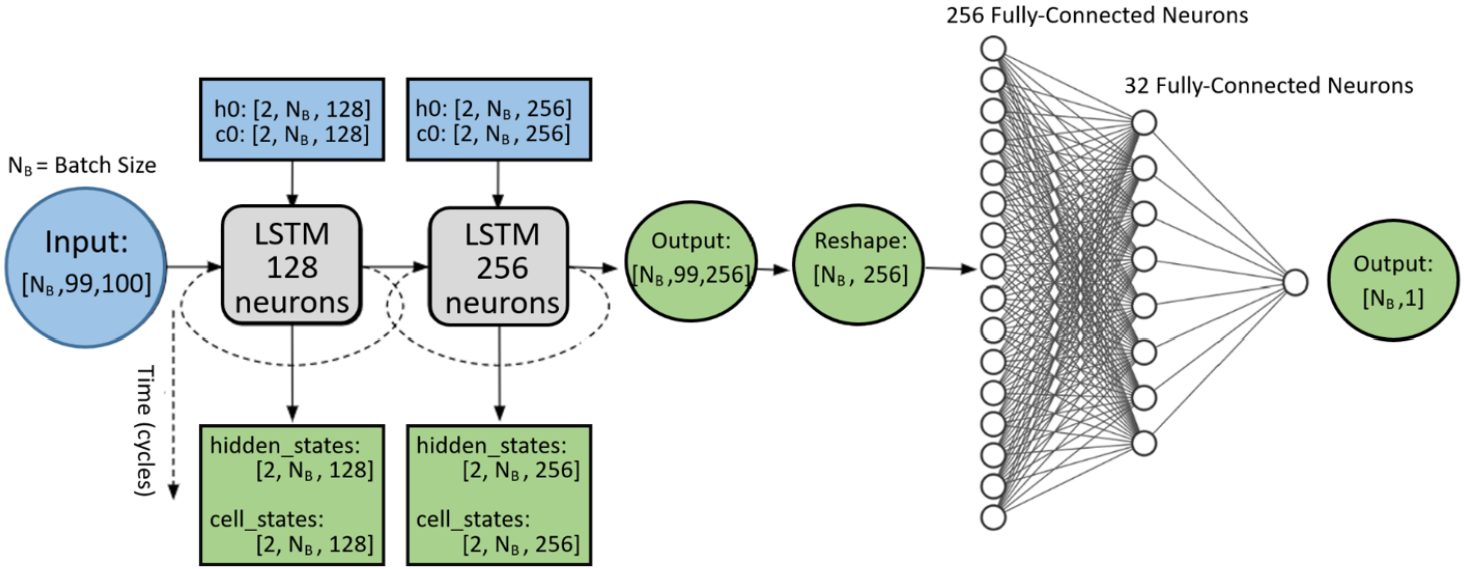


Fig. 5. Architecture of Long-Short-Term-Memory Recurrent Neural network

LSTM

- Pytorch
  - Dataset
  - Dataloader

```python
from torch.utils.data import Dataset
class CustomDataset(Dataset):
    def __init__(self, features, targets):
        self.features = features
        self.targets = targets

    def __getitem__(self, idx):
        return self.features[idx], self.targets[idx]

    def __len__(self):
        return len(self.features)
```

```python
train_Dataset = CustomDataset(train_Qdlin,y_train)
test_Dataset = CustomDataset(test_Qdlin,y_test)
secondary_test_Dataset = CustomDataset(secondary_test_Qdlin,y_sec)

train_Qdlin_loader = DataLoader(train_Dataset, batch_size=41, shuffle=True)
test_Qdlin_loader = DataLoader(test_Dataset, batch_size=42, shuffle=True)
secondary_test_Qdlin_loader = DataLoader(secondary_test_Dataset, batch_size=40, shuffle=True)
```

LSTM

- Pytorch
  - torch.nn.LSTM

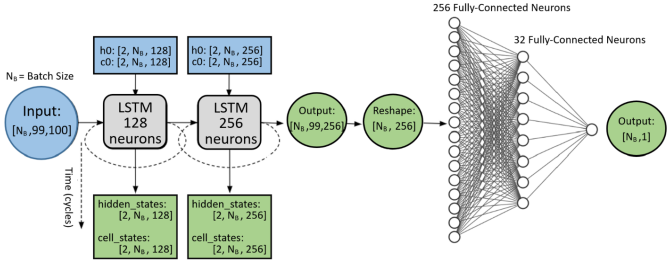


Fig. 5. Architecture of Long-Short-Term-Memory
Recurrent Neural network

```
class LSTMRNN(nn.Module):

    def __init__(self, input_size, hidden_size, hidden_size2, num_layers,
                 dropout, num_dense_neurons):

        super(LSTMRNN, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.hidden_size2 = hidden_size2
        self.num_layers = num_layers
        self.dropout = dropout
        self.num_dense_neurons = num_dense_neurons
        self.firstEpochFlag = True

        self.dropout_layer = nn.Dropout(self.dropout)

        self.lstm = nn.LSTM(input_size = self.input_size,
                            hidden_size = self.hidden_size,
                            num_layers = self.num_layers,
#                             dropout = self.dropout,
                            batch_first = True)

        self.lstm2 = nn.LSTM(input_size = self.hidden_size,
                            hidden_size = self.hidden_size2,
                            num_layers = self.num_layers,
#                             dropout = self.dropout,
                            batch_first = True)

        self.linear_layer = nn.Sequential(
            #nn.Dropout(self.dropout)
            nn.Linear(self.hidden_size2, self.num_dense_neurons),
            nn.ReLU(),
            nn.Linear(num_dense_neurons,1),
        )

    def forward(self, x):
        if(self.firstEpochFlag):print(x.shape)
        h0 = Variable(torch.zeros(self.num_layers, x.shape[0], self.hidden_size,device=x.device))
        c0 = Variable(torch.zeros(self.num_layers, x.shape[0], self.hidden_size,device=x.device))
        lstm_output,(h1, c1) = self.lstm(x,(h0, c0))
        if(self.firstEpochFlag):print(lstm_output.shape)

        h00 = Variable(torch.zeros(self.num_layers, x.shape[0], self.hidden_size2,device=x.device))
        c00 = Variable(torch.zeros(self.num_layers, x.shape[0], self.hidden_size2,device=x.device))
        lstm_output2,(h2, c2) = self.lstm2(lstm_output,(h00, c00))
        if(self.firstEpochFlag):print(lstm_output2.shape)

        flat = lstm_output2[:, -1, :]#last projected timestep (cycle)
        if(self.firstEpochFlag):print(flat.shape)

        output = self.linear_layer(flat)
```

LSTM

- Train
- Test

```
def train(model,train_loader,criterion,optimizer,epochs,device):
    for epoch in range(epochs):
        model.train()
        for inputs, targets in train_loader:
            optimizer.zero_grad()
            inputs = inputs.to(device)
            targets = targets.to(device)

            outputs = model(inputs)
            outputs.to(device)
            loss = criterion(outputs, targets)
            loss.backward()
            optimizer.step()
        model.eval()
        if(epoch%100 == 0):
            print('--',epoch,'-epoch--RMSE-- ',
                  np.sqrt(mean_squared_error(10**(targets.cpu().detach().numpy()+y_train_mean.numpy()),
                                             10**(outputs.cpu().detach().numpy()+y_train_mean.numpy()))))
    return model
```

```
criterion=nn.MSELoss()
model=LSTMRNN(input_size, hidden_size, hidden_size2, num_layers,dropout,
optimizer=optim.Adam(model.parameters(),weight_decay=weight_decay, lr=lr)
model = model.to(device)
model = train(model,train_Qdlin_loader,criterion,optimizer,epochs,device)
```

```
torch.Size([41, 99, 100])
torch.Size([41, 99, 128])
torch.Size([41, 99, 256])
torch.Size([41, 256])
-- 0 -epoch--RMSE--  325.713504214364
-- 100 -epoch--RMSE--  191.60093282624348
-- 200 -epoch--RMSE--  118.58940215229036
-- 300 -epoch--RMSE--  84.2295624764314
-- 400 -epoch--RMSE--  65.92709414591444
-- 500 -epoch--RMSE--  52.18300708080519
-- 600 -epoch--RMSE--  46.43477470967451
-- 700 -epoch--RMSE--  43.97749221237031
-- 800 -epoch--RMSE--  42.366674639828105
-- 900 -epoch--RMSE--  38.88688856498636
-- 1000 -epoch--RMSE--  37.52256793585603
-- 1100 -epoch--RMSE--  36.52446165852783
-- 1200 -epoch--RMSE--  35.78776480837691
-- 1300 -epoch--RMSE--  35.32158666670042
-- 1400 -epoch--RMSE--  34.85211871891183
```

Convolutional Neural Network (CNN)

Convolution layers

Max Pool layers

Fully connected layers

- Discharge capacity-voltage data
  - Two-dimensional (99X100)
- Tabular summary data (discharge features)
  - 6 features

Convolutional Neural Network (CNN)

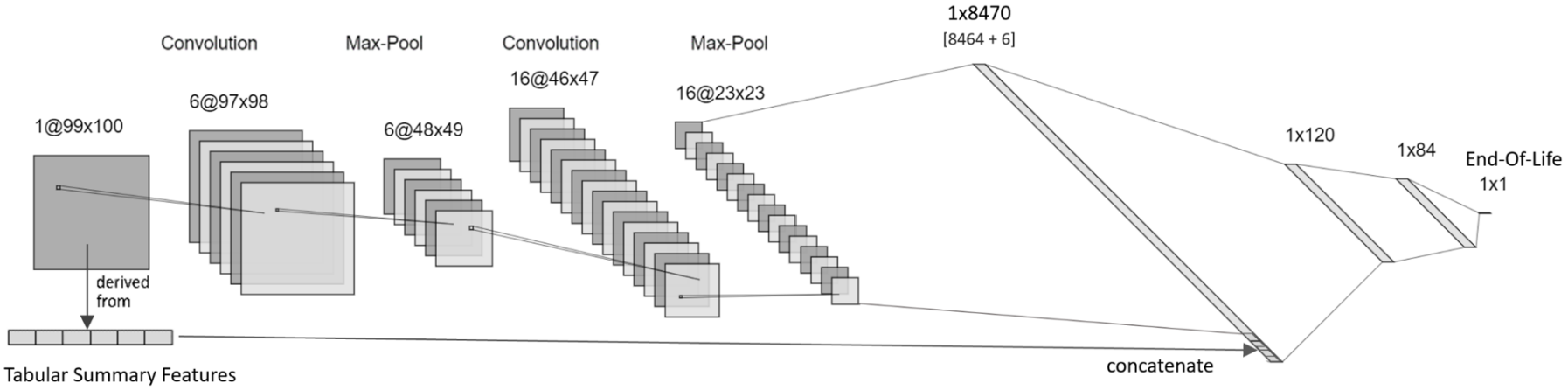


Fig. 7. Architecture of Convolutional Neural network

CNN

- Pytorch

  torch.nn.Conv2d

```
from torch.utils.data import Dataset
class CustomDataset(Dataset):
    def __init__(self, features, targets, tabular):
        self.features = features
        self.targets = targets
        self.tabular = tabular #Merge tabular data set

    def __getitem__(self, idx):
        return self.features[idx], self.targets[idx], self.tabular[idx]

    def __len__(self):
        return len(self.features)
```

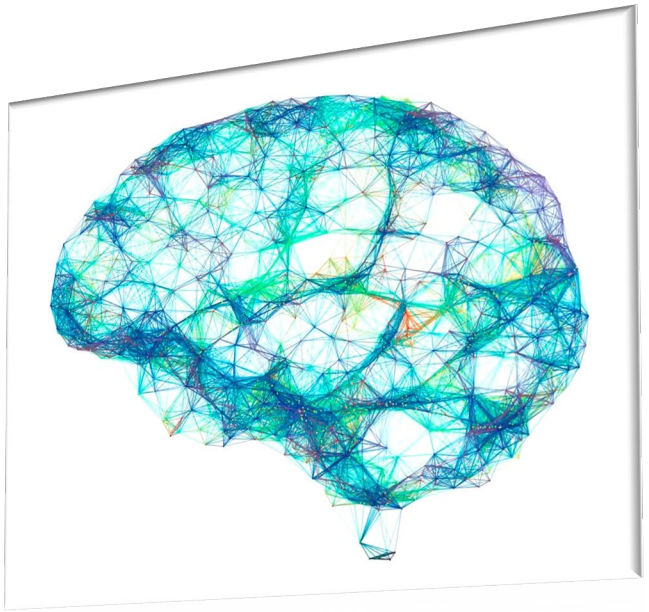
```
class CNN(nn.Module):

    def __init__(self,inputSize):

        super(CNN, self).__init__()
        self.firstEpochFlag=True
        # Calculates size of first linear layer
        # This allows the Module to be used with different number of cycles
        self.linearSize=((((inputSize-2)//2)-2)//2)*23*16+6

        # Convolutional architecture from Attia et al.
        self.convolutional_layer = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=6, kernel_size=(3,3)),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            #nn.BatchNorm2d(3),
            nn.Conv2d(in_channels=6, out_channels=16, kernel_size=(3,3)),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2, padding=0),
        )
        self.linear_layer = nn.Sequential(
            nn.Linear(self.linearSize,120),
            #nn.Dropout(.1)
            nn.ReLU(),
            nn.Linear(120,84),
            nn.ReLU(),
            nn.Linear(84,1),
        )

    def forward(self, x,tabular):
        #Print the shapes of the data on the first Epoch
        if(self.firstEpochFlag):print(x.shape)

        x = x.view(-1, 1,x.shape[1],99)
        if(self.firstEpochFlag):print(x.shape)

        clout = self.convolutional_layer(x)
        if(self.firstEpochFlag):print(clout.shape)

        flat = torch.flatten(clout, 1)
        if(self.firstEpochFlag):print(flat.shape)

        #Concatenate tabular features!
        flat=torch.cat((flat, tabular), -1)
        if(self.firstEpochFlag):
            print(tabular.shape)
            print(flat.shape)
            self.firstEpochFlag=False

        output = self.linear_layer(flat)
        return output
```

Results & Conclusion

- RMSE & MPE

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\widehat{y_i} - y_i)^2}$$

$$MPE = \frac{100\%}{n}\sum_{i=1}^{n}\left(\frac{\widehat{y_i} - y_i}{y_i}\right)$$



Summary data results

TABLE II
RESULTS FROM TABULAR SUMMARY DATA

MODEL	RMSE TRAIN	RMSE TEST	RMSE TEST #2	MPE TRAIN	MPE TEST	MPE TEST #2
Severson et al. Elastic Net "Variance" model	(103)	(138)	(196)	(14)	(13)	(11)
"Discharge" model	(76)	(86)	(173)	(10)	(10)	(9)
"Full" model	(51)	(100)	(214)	(6)	(8)	(11)
Support Vector Regression - "Discharge" features	79	79	172	10	10	9
Support Vector Regression - "Full" features	54	115	200	7	9	11
Multilayer Perceptron - "Discharge" features	**67**	**105**	**159**	**8**	**8**	**8**
Multilayer Perceptron - "Full" features	39	138	176	4	8	10

Discharge capacity-voltage data results

TABLE III

RESULTS FROM DISCHARGE CAPACITY-VOLTAGE DATA

MODEL	RMSE TRAIN	RMSE TEST	RMSE TEST #2	MPE TRAIN	MPE TEST	MPE TEST #2
Xu et al. [9] LSTM 100 cycles	(42)	(91)	(215)			
60 cycles	(48)	(88)	(212)			
40 cycles	(51)	(127)	(224)			
LSTM 100 cycles	35	101	193	1	11	14
60 cycles	38	316	220	2	18	17
40 cycles	49	127	218	3	12	17
Attia et al. [8] Convolutional Neural Network	(17)	(72)	(204)			
Convolutional neural network 100 cycles	**16**	**83**	**141**	**1**	**6**	**8**
60 cycles	14	184	203	1	8	12
40 cycles	11	80	206	1	6	11

Xu, P., & Lu, Y. (2022). Predicting Li-ion Battery Cycle Life with LSTM RNN. arXiv preprint arxiv.org/abs/2207.03687

Attia, P. M., Severson, K. A., & Witmer, J. D. (2021). Statistical learning for accurate and interpretable battery lifetime prediction. *Journal of The Electrochemical Society*, *168*(9), 090547. doi.org/10.1149/1945-7111/ac2704

Discharge capacity-voltage data results

- Convolutional neural network (100 cycles)
  - Best model

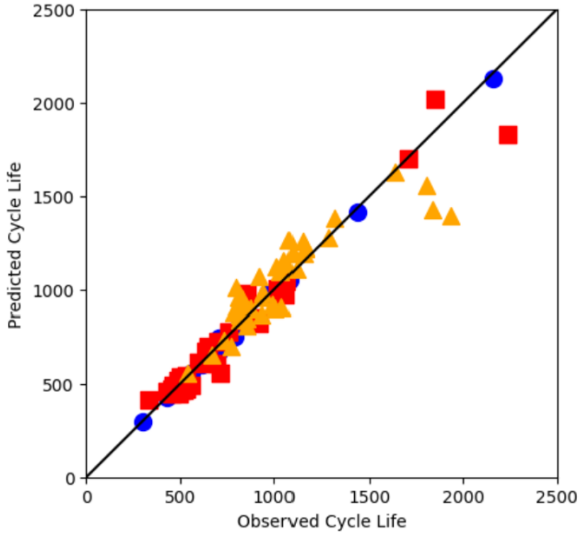


Fig. 8. Predicted vs Observed End-Of-Life
CNN - first 100 cycles

KANSAS STATE UNIVERSITY

Discussion & Future work

- Unbalanced (few long-life batteries)
  - Oversampling?
  - Artificial data?
  - Repeat using Odd cycles/even cycles?
- Overfitting
  - Regularization
- HyperParameter search
- Cross-validation LSTM/CNN
  - Tensorflow?

Discussion & Future work

- Use more data
  - Voltage
  - Current
  - Temperature
  - Internal resistance
  - Trade-off
    - Capacity data is “free”

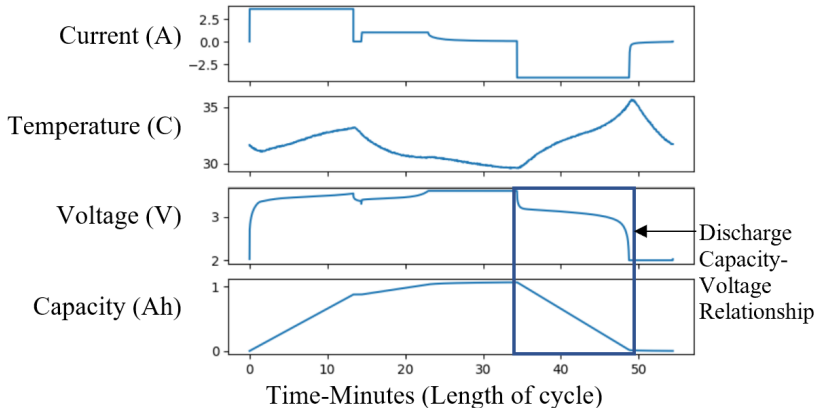


Fig. 2. Timeseries Data ,
Sample - cycle 1 of battery ‘b1c0’

- Pre-trained models CNN &Transformers

Machine Learning Methods for Early Prediction of End-of-Life for Lithium-ion Batteries

James Chapman
CIS 732 Machine Learning – Term Project
Kansas State University
jachapman@ksu.edu

CLASStorch.nn.Conv2d

- [Pytorch website]

Variables:

- **weight** (*Tensor*) – the learnable weights of the module of shape $(\text{out\_channels}, \frac{\text{in\_channels}}{\text{groups}}, \text{kernel\_size}[0], \text{kernel\_size}[1])$. The values of these weights are sampled from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{groups}{C_{\text{in}} * \prod_{i=0}^{1} \text{kernel\_size}[i]}$
- **bias** (*Tensor*) – the learnable bias of the module of shape (out_channels). If `bias` is `True`, then the values of these weights are sampled from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{groups}{C_{\text{in}} * \prod_{i=0}^{1} \text{kernel\_size}[i]}$