Requirements

Calling tensorflow, matplotlib, pandas, numpy and other libraries for prediction and visualization aid.

```
%tensorflow_version 2.x
%matplotlib inline
!pip show tensorflow
```

```
    Name: tensorflow
    Version: 2.5.0
    Summary: TensorFlow is an open source machine learning framework for everyone.
    Home-page: https://www.tensorflow.org/
    Author: Google Inc.
    Author-email: packages@tensorflow.org
    License: Apache 2.0
    Location: /usr/local/lib/python3.7/dist-packages
    Requires: astunparse, absl-py, keras-nightly, wrapt, numpy, typing-extensions, tensorflow-estimator, termcolor, gast,
    Required-by: kapre
```

```
import datetime
import numpy as np
import pandas as pd
from scipy.io import loadmat
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
from sklearn import metrics
import matplotlib.pyplot as plt
import seaborn as sns
```

Function for making MAT data usable for prediction

The data that we used here is a part of NASA's Prognostics Center of Excellence Repository. It can be accessed at the following link: https://ti.arc.nasa.gov/tech/dash/groups/pcoe/prognostic-data-repository/#battery The data files are in .mat format. We required a function that can convert these highly branched .mat data to simpler dataframe that can be used for learning the patterns from.

```
def load_data(battery):
  mat = loadmat(battery + '.mat')
  print('Total data in dataset: ', len(mat[battery][0, 0]['cycle'][0]))
  capacity_ = []
  counter = 0
  for i in range(len(mat[battery][0, 0]['cycle'][0])):
    row = mat[battery][0, 0]['cycle'][0, i]
    if row['type'][0] == 'discharge':
      ambient_temperature = row['ambient_temperature'][0][0]
      date_time = datetime.datetime(int(row['time'][0][0]),
                                   int(row['time'][0][1]),
                                   int(row['time'][0][2]),
                                   int(row['time'][0][3]),
                                   int(row['time'][0][4])) + datetime.timedelta(seconds=int(row['time'][0][5]))
      data = row['data']
      capacity = data[0][0]['Capacity'][0][0]
      capacity_.append([counter + 1, ambient_temperature, date_time, capacity])
      counter = counter + 1
  return pd.DataFrame(data=capacity_,
                      columns=['cycle', 'ambient_temperature', 'datetime',
                               'capacity'])
```

```
df = load_data('B0007')
```

```
Total data in dataset:  616
```

For the task of training RNN, we need data cycles that have capacity attribute in them. Only "Discharge" cycles have capacity attribute in them. So, Out of 616 total cycles including Charge, Discharge and Impedance, only those of type Discharge are going to be used here.

```
df
```

	cycle	ambient_temperature	datetime	capacity
0	1	24	2008-04-02 15:25:41	1.891052
1	2	24	2008-04-02 19:43:48	1.880637
2	3	24	2008-04-03 00:01:06	1.880663
3	4	24	2008-04-03 04:16:37	1.880771
4	5	24	2008-04-03 08:33:25	1.879451
...	...	...	...	...
163	164	24	2008-05-26 10:44:38	1.406171
164	165	24	2008-05-26 15:30:43	1.406336
165	166	24	2008-05-26 20:21:04	1.400455
166	167	24	2008-05-27 15:52:41	1.421787
167	168	24	2008-05-27 20:45:42	1.432455

168 rows × 4 columns

Now we used this simpler dataframe in order to prepare training and testing sets for validation of the model in further steps. Here we intend to use an LSTM model, so only getting the temporal series of capacity data with cycle sequence can also get us results. In this specific case, the battery capacity data is used to predict the capacity in the following cycles using the data of the first few cycles in such a way that we can know when the battery threshold is reached and estimate the missing cycles to reach the end of the battery life.

```
attrib=['cycle', 'datetime', 'capacity']
dis_ele = df[attrib]
rows=['cycle','capacity']
dataset=dis_ele[rows]
dataset
```

	cycle	capacity
0	1	1.891052
1	2	1.880637
2	3	1.880663
3	4	1.880771
4	5	1.879451
...	...	...
163	164	1.406171
164	165	1.406336
165	166	1.400455
166	167	1.421787
167	168	1.432455

168 rows × 2 columns

Now We need to define the Train-Test Split: Out of the 168 cycles we can train model on some of the earlier capacity-cycle data and then check the results on remaining capacity-cycle data as our test data.

We shall try it for a range of values from using a train test split of 50-118, 55-113, and so on untill 75-83. We shall store them in a dictionary for each split case wise.

Here train is intentioanlly smaller subset because we want to take a part of battery data and predict for its failure in future as long as possible. Here failure means Battery capacity below 80%.

```
sc=MinMaxScaler(feature_range=(0,1))
data_dict = {}
for i in range(50, 80, 5):
    data_train = dataset[(dataset['cycle']<i)]
    data_set_train = data_train.iloc[:,1:2].values
    data_test = dataset[(dataset['cycle']>=i)]
    data_set_test = data_test.iloc[:,1:2].values
    data_set_train=sc.fit_transform(data_set_train)
    data_set_test=sc.transform(data_set_test)
    data_dict['data_train' + str(i)], data_dict['data_set_train' + str(i)], data_dict['data_test' + str(i)], data_dict['data
```

```
data_dict['data_set_test75']
```

```
           [-0.45079162],
           [-0.45257237],
           [-0.49569504],
           [-0.51895693],
           [-0.5182625 ],
           [-0.51756809],
           [-0.5393099 ],
           [-0.56127255],
           [-0.5823167 ],
           [-0.58249785],
           [-0.58106379],
           [-0.49715723],
           [-0.47545046],
           [-0.5404886 ],
           [-0.58259728],
           [-0.60689874],
           [-0.62613876],
           [-0.64867001],
           [-0.66952217],
           [-0.67134238],
           [-0.69201882],
           [-0.71276241],
           [-0.71170468],
           [-0.73609009],
           [-0.71541884],
           [-0.64735117],
           [-0.71468107],
           [-0.73548488],
           [-0.75660842],
           [-0.75589259],
           [-0.77785086],
           [-0.79983459],
                [-0.82109955],
                [-0.84348944],
                [-0.84239536],
                [-0.86251913],
                [-0.8627668 ],
                [-0.8853119 ],
                [-0.90683739],
                [-0.90444847],
                [-0.90663241],
                [-0.88330873],
                [-0.77703044],
                [-0.84135512],
                [-0.86116976],
                [-0.8835894 ],
                [-0.92618504],
                [-0.92544094],
                [-0.94871914],
                [-0.94942562],
                [-0.96965256],
                [-0.99034772],
                [-0.98929575],
                [-1.01335312],
                [-1.01270415],
                [-1.03292745],
                [-1.0322381 ],
                [-1.05689333],
                [-0.96745917],
```

Now we need to convert this train split into time sequence data that is important for LSTM. What we shall do is, for each capacity value use past few capacity values as its features. That way X_train will be array of past few capacities and y_train will be array of current capacity.

Here we again allow performance to decide how many previous capacity values should be features in X_train for current capacity value in y_train.

```
train = {}
#take the last 10t to predict 10t+1
for k in range(50, 80, 5):
  for j in range(10, 30, 5):
    X_train=[]
    X_train=[]
    y_train=[]
    for i in range(j,k-1):
      X_train.append(data_dict['data_set_train'+ str(k)] [i-j:i,0])
      y_train.append(data_dict['data_set_train'+ str(k)][i,0])
    X_train,y_train=np.array(X_train),np.array(y_train)
    X_train=np.reshape(X_train,(X_train.shape[0],X_train.shape[1],1))
    train['X_train' + str(j) + '_' + str(k)], train['y_train' + str(j) + '_' + str(k)] = X_train, y_train
```

Now there is train data corresponding to each split (50, 55, ..,75) and each selection of features (10, 15, .., 25).

```
train['X_train25_75']
```

```
    array([[[1.        ],
            [0.95633261],
            [0.95644012],
            ...,
            [0.91596567],
            [0.91257195],
            [0.91455096]],

           [[0.95633261],
            [0.95644012],
            [0.95689389],
            ...,
            [0.91257195],
            [0.91455096],
            [0.86818433]],

           [[0.95644012],
            [0.95689389],
            [0.95135949],
            ...,
            [0.91455096],
            [0.86818433],
            [0.8699513 ]],

           ...,

           [[0.5357689 ],
                 [0.68145799],
                 [0.68408604],
                 ...,
                 [0.10644394],
                 [0.08665058],
                 [0.06246199]],

                [[0.68145799],
                 [0.68408604],
                 [0.61927088],
                 ...,
                 [0.08665058],
                 [0.06246199],
                 [0.04078416]],

                [[0.68408604],
                 [0.61927088],
                 [0.57820153],
                 ...,
                 [0.06246199],
                 [0.04078416],
                 [0.02048362]]])
```

This does result in number of rows reduced by number of features but now we have sequential data of capacities.

Neural Network building

Calling the necessary tensorflow models, layers and optimizers.

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import LSTM
from tensorflow.keras.optimizers import Adam
```

Here we used LSTM(long short term memory) artificial recurrent neural network architecture. Its difference from the standard neural networks is that it has feedback capabilities unlike their feedforward connections. Most of the RNN's are capable of using feedback or memory advantage to optimize their learning but a standard RNN may find it difficult to utilize a very distant learning in the network. This is called **Long Term Dependency issue**.

A Long Short Term Memory (**LSTM**) RNN is capable of dealing with long term dependencies as it is innate nature of its layer architecture. It has a bit more complex layer structure than a standard RNN which helps it become more selective at decision making and learning from data.

Let us now define different models for different train sets since input size differes for each of them. We can again save them in a dictionary.

```
model = {}
for i in range(50, 80, 5):
  for j in range(10, 30, 5):
    regress = Sequential()
    regress.add(LSTM(units=300, return_sequences=True, input_shape=(train['X_train'+str(j)+'_'+str(i)].shape[1],1)))
    regress.add(Dropout(0.3))
    regress.add(LSTM(units=300, return_sequences=True))
    regress.add(Dropout(0.3))
    regress.add(LSTM(units=300, return_sequences=True))
    regress.add(Dropout(0.3))
    regress.add(LSTM(units=300))
    regress.add(Dropout(0.3))
    regress.add(Dense(units=1))
    regress.compile(optimizer='adam',loss='mean_squared_error')
    model['regress'+str(j)+'_'+str(i)] = regress
```

```
model['regress10_50']
```

```
<tensorflow.python.keras.engine.sequential.Sequential at 0x7f3d3610f810>
```

Train

Now here we have 24 models to train on 24 sets of X-train and y_train.

```
for i in range(50, 80, 5):
  for j in range(10, 30, 5):
    model['regress'+str(j)+'_'+str(i)].fit(train['X_train'+str(j)+'_'+str(i)],train['y_train'+str(j)+'_'+str(i)],epochs=22
```

```
2/2 [==============================] - 1s 288ms/step - loss: 0.0058
Epoch 91/220
2/2 [==============================] - 1s 289ms/step - loss: 0.0043
Epoch 92/220
2/2 [==============================] - 1s 289ms/step - loss: 0.0055
Epoch 93/220
2/2 [==============================] - 1s 292ms/step - loss: 0.0057
Epoch 94/220
2/2 [==============================] - 1s 278ms/step - loss: 0.0050
Epoch 95/220
2/2 [==============================] - 1s 292ms/step - loss: 0.0051
Epoch 96/220
2/2 [==============================] - 1s 278ms/step - loss: 0.0066
Epoch 97/220
2/2 [==============================] - 1s 301ms/step - loss: 0.0047
Epoch 98/220
2/2 [==============================] - 1s 286ms/step - loss: 0.0058
Epoch 99/220
2/2 [==============================] - 1s 276ms/step - loss: 0.0053
Epoch 100/220

2/2 [==============================] - 1s 287ms/step - loss: 0.0056
Epoch 101/220
2/2 [==============================] - 1s 271ms/step - loss: 0.0066
Epoch 102/220
2/2 [==============================] - 1s 296ms/step - loss: 0.0034
Epoch 103/220
2/2 [==============================] - 1s 291ms/step - loss: 0.0059
Epoch 104/220
2/2 [==============================] - 1s 277ms/step - loss: 0.0049
Epoch 105/220
2/2 [==============================] - 1s 278ms/step - loss: 0.0056
```

```
Epoch 106/220
2/2 [==============================] - 1s 282ms/step - loss: 0.0068
Epoch 107/220
2/2 [==============================] - 1s 275ms/step - loss: 0.0061
Epoch 108/220
2/2 [==============================] - 1s 270ms/step - loss: 0.0063
Epoch 109/220
2/2 [==============================] - 1s 289ms/step - loss: 0.0065
Epoch 110/220
2/2 [==============================] - 1s 274ms/step - loss: 0.0041
Epoch 111/220
2/2 [==============================] - 1s 280ms/step - loss: 0.0054
Epoch 112/220
2/2 [==============================] - 1s 285ms/step - loss: 0.0054
Epoch 113/220
2/2 [==============================] - 1s 308ms/step - loss: 0.0053
Epoch 114/220
2/2 [==============================] - 1s 294ms/step - loss: 0.0045
Epoch 115/220
2/2 [==============================] - 1s 303ms/step - loss: 0.0055
Epoch 116/220
2/2 [==============================] - 1s 288ms/step - loss: 0.0052
Epoch 117/220
2/2 [==============================] - 1s 299ms/step - loss: 0.0043
Epoch 118/220
2/2 [==============================] - 1s 279ms/step - loss: 0.0060
Epoch 119/220
2/2 [==============================] - 1s 295ms/step - loss: 0.0052
```

Test prep

Preparing the test data.

Corresponding to each split and feature count selection we have different test sets. We shall first take different subsets (inputs) of total 168 row initial capacity data due to different splits.

```
Inputs = {}
```

```
for i in range(50, 80, 5):
  data_total = pd.concat((data_dict['data_train'+str(i)]['capacity'], data_dict['data_test'+str(i)]['capacity']),axis=0)
  for j in range(10, 30, 5):
    inputs=data_total[len(data_total)-len(data_dict['data_test'+str(i)])-j:].values
    inputs=inputs.reshape(-1,1)
    inputs=sc.transform(inputs)
    Inputs['inputs'+str(j)+'_'+str(i)] = inputs
```

```
Inputs['inputs15_50'].shape
```

```
     (134, 1)
```

Testing

Now in the test dictionary we can add different test datasets corresponding to different split and feature count combinations.

```
test = {}
for k in range(50, 80, 5):
  for j in range(10, 30, 5):
    X_test=[]
    for i in range(j,len(Inputs['inputs'+str(j)+'_'+str(k)])):
      X_test.append(Inputs['inputs'+str(j)+'_'+str(k)][i-j:i,0])
    X_test=np.array(X_test)
    X_test=np.reshape(X_test,(X_test.shape[0],X_test.shape[1],1))
    test['test'+str(j)+'_'+str(k)] = X_test
```

Now we can get the performance results in terms of Root Mean Square Error and R2 Score for each test set and model combination. The one with best reults will be plotte later to check its performance visually.

```
for i in range(50, 80, 5):
  for j in range(10, 30, 5):
    pred=model['regress'+str(j)+'_'+str(i)].predict(test['test'+str(j)+'_'+str(i)])
    print(pred.shape)
    pred=sc.inverse_transform(pred)
    pred=pred[:,0]
    tests=data_dict['data_test'+str(i)].iloc[:,1:2]
    print('Performance of model with {} time features and {} rowed training instances:'.format(str(j), str(i)))
    rmse = np.sqrt(mean_squared_error(tests, pred))
    print('Test RMSE: %.3f' % rmse)
    r2 = metrics.r2_score(tests,pred)
    print('Test r2 score: %.3f' % r2)
    print('\n')
```

```
(104, 1)
Performance of model with 20 time features and 65 rowed training instances:
Test RMSE: 0.144
Test r2 score: -2.009


(104, 1)
Performance of model with 25 time features and 65 rowed training instances:
Test RMSE: 0.274
Test r2 score: -9.856


(99, 1)
Performance of model with 10 time features and 70 rowed training instances:
Test RMSE: 0.058
Test r2 score: 0.436


(99, 1)
Performance of model with 15 time features and 70 rowed training instances:
Test RMSE: 0.059
Test r2 score: 0.425


(99, 1)
Performance of model with 20 time features and 70 rowed training instances:
Test RMSE: 0.061
Test r2 score: 0.368


(99, 1)
Performance of model with 25 time features and 70 rowed training instances:
Test RMSE: 0.095
Test r2 score: -0.519


(94, 1)
Performance of model with 10 time features and 75 rowed training instances:
Test RMSE: 0.020
Test r2 score: 0.926


(94, 1)
Performance of model with 15 time features and 75 rowed training instances:
Test RMSE: 0.033
Test r2 score: 0.790


(94, 1)
Performance of model with 20 time features and 75 rowed training instances:
Test RMSE: 0.030
Test r2 score: 0.832


(94, 1)
Performance of model with 25 time features and 75 rowed training instances:
Test RMSE: 0.028
Test r2 score: 0.849
```

Since, best performance is shown by the hyperparameter selection of 55 train examples split and 15 training data features.

```
X_test=[]
for i in range(15,len(Inputs['inputs15_55'])):
    X_test.append(Inputs['inputs15_55'][i-15:i,0])
X_test=np.array(X_test)
X_test=np.reshape(X_test,(X_test.shape[0],X_test.shape[1],1))


pred=model['regress15_55'].predict(X_test)
print(pred.shape)
pred=sc.inverse_transform(pred)
pred=pred[:,0]
```

```
tests=data_dict['data_test55'].iloc[:,1:2]
rmse = np.sqrt(mean_squared_error(tests, pred))
print('Test RMSE: %.3f' % rmse)
r2 = metrics.r2_score(tests,pred)
print('Test r2 score: %.3f' % r2)
```

```
(114, 1)
Test RMSE: 0.016
Test r2 score: 0.974
```

The model was able to predict the remaining useful life for battery test dataset with a root mean square error of 0.016 cycles and the R2 score is attained is reasonably well 97.4%.

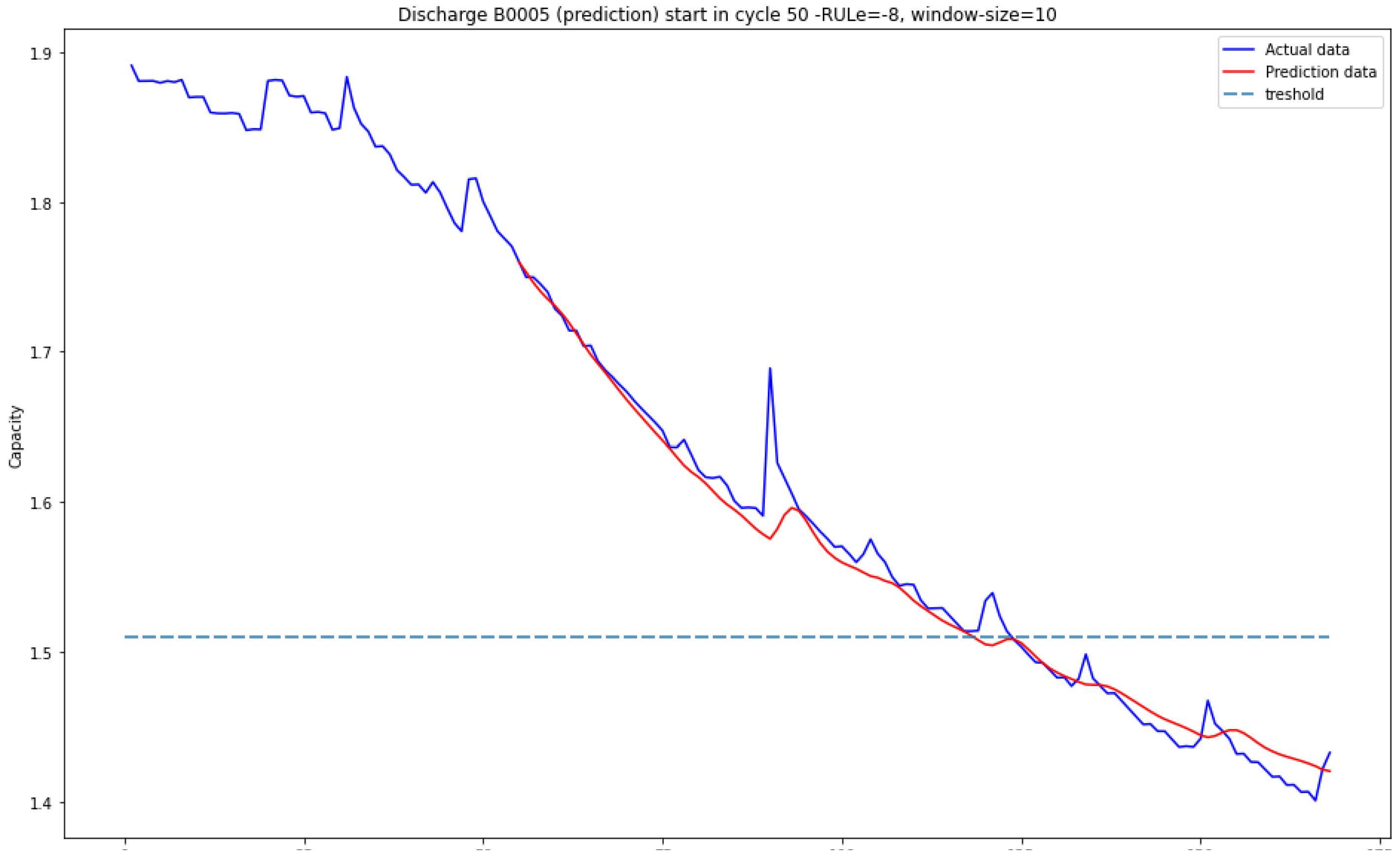
Visual rep of results

```
ln = len(data_dict['data_train55'])
data_dict['data_test55']['pre']=pred
plot_df = dataset.loc[(dataset['cycle']>=1),['cycle','capacity']]
plot_pre = data_dict['data_test55'].loc[(data_dict['data_test55']['cycle']>=ln),['cycle','pre']]
plt.figure(figsize=(16, 10))
plt.plot(plot_df['cycle'], plot_df['capacity'], label="Actual data", color='blue')
plt.plot(plot_pre['cycle'],plot_pre['pre'],label="Prediction data", color='red')
#Draw threshold
plt.plot([0.,168], [1.51, 1.51],dashes=[6, 2], label="treshold")
plt.ylabel('Capacity')
# make x-axis ticks legible
adf = plt.gca().get_xaxis().get_major_formatter()
plt.xlabel('cycle')
plt.legend()
plt.title('Discharge B0005 (prediction) start in cycle 50 -RULe=-8, window-size=10')
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-

Text(0.5, 1.0, 'Discharge B0005 (prediction) start in cycle 50 -RULe=-8, window-size=10')
```



Prediction on the Test set and error calculation

```
pred=0
Afil=0
Pfil=0
a=data_dict['data_test55']['capacity'].values
b=data_dict['data_test55']['pre'].values
j=0
k=0
for i in range(len(a)):
    actual=a[i]

    if actual<=1.51:
        j=i
        Afil=j
        break
for i in range(len(a)):
    pred=b[i]
    if pred< 1.51:
        k=i
        Pfil=k
        break
print("The Actual fail at cycle number: "+ str(Afil+ln))
print("The prediction fail at cycle number: "+ str(Pfil+ln))
RULerror=Pfil-Afil
print("The error of RUL= "+ str(RULerror)+ " Cycle(s)")
```

```
The Actual fail at cycle number: 123
The prediction fail at cycle number: 118
The error of RUL= -5 Cycle(s)
```

We have assumed the End of Life for a battery to be when the State of Charge goes below 80%, which is 1.51 Ahr for the battery used here.

On the test data battery, this model was able to predict the failure(State of charge hitting 20%) with an error of 5 cycles. The fact to be observed is that the model was capable to pre-predict the failure(End of Life) for battery which is very important to avoid complications and improve the quality of Predictive Maintenace feature for this (RUL)Remaining Useful Life Prediction Task.

✓ 0s completed at 12:02 AM