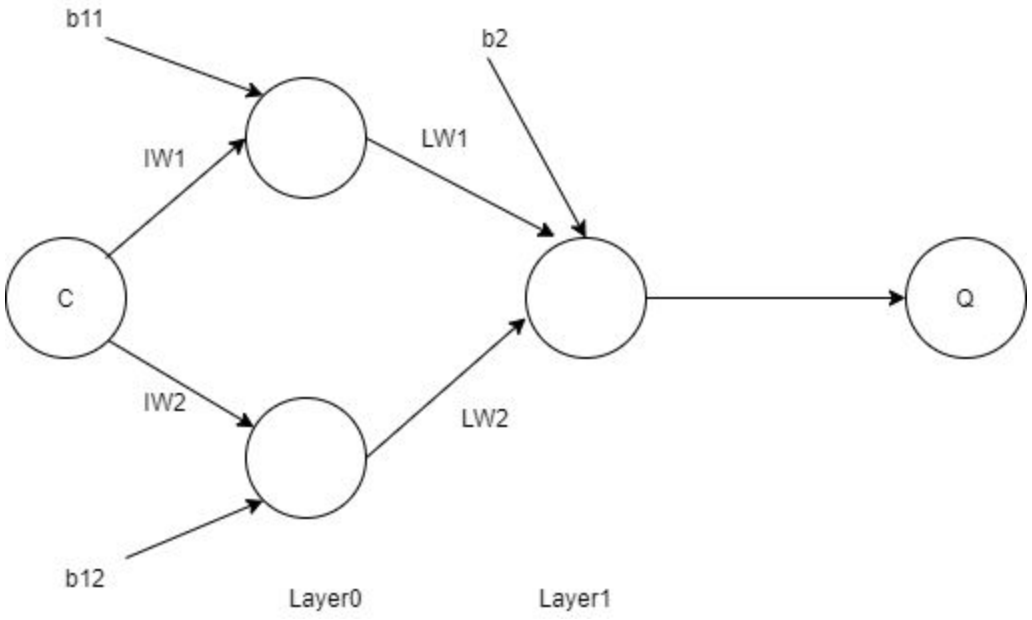
Research Paper Title :

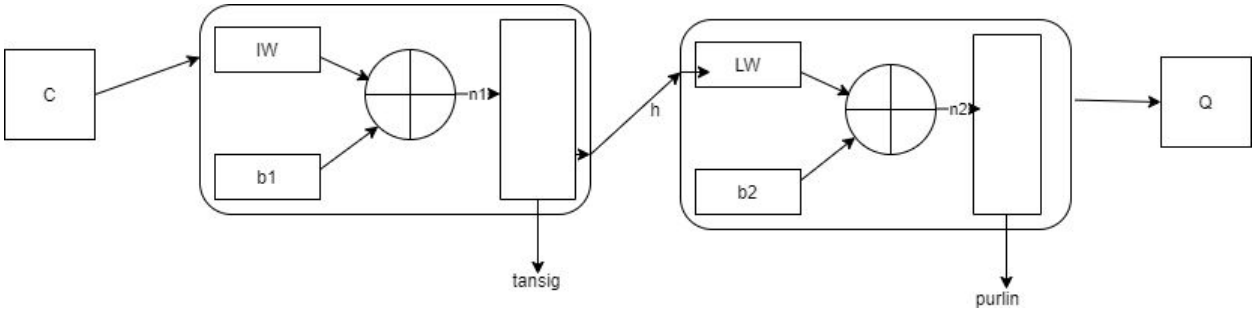
Remaining Useful Life Prediction of Lithium-Ion Batteries Using Neural Network and Bat-Based Particle Filter

Research Paper Implementation :

The architecture of the neural network used in this paper is as shown :



This is a neural network model of 2 neurons where,
IW1, IW2 - Weights for 2 neurons of layer 0
LW1, LW2 - Weights for neuron of layer 1
b11, b12, b2 - bias units
Q - Output (capacity)



Activation functions used in the neural network model : Tansig and Purlin function.

Tansig is the tanh function used in python while Purlin function is a neural transfer function which is linear.
Example : a = purelin(n) = n

The NN architecture consists of 3 layer(one input layer, one output layer and one hidden layer)
The input layer has only a single input (number of cycles).
The output layer has single output(the predicted capacity of the battery for the cycle number using purlin/linear activation function)
The hidden layer the neural network uses two neurons with tansig as activation function.
Particle filters using the Bat optimization algorithm are used as optimizers to adjust the parameters (i.e. weights and biases) of the neural network.

Summary of Neural Network :

Input layer	(single input)
A single hidden layer	(two neurons with tansig as activation function)
A output layer	(single neuron with purlin/linear activation function).
Input to the neural network	Cycle No.
Output of the network	Capacity
Optimizer:	Particle filter using Bat optimization algorithm

Code:

Neural Network using Bat PF Optimizer

```
“””Bat PF optimizer”””
import math
from keras.optimizers import Optimizer
from keras import backend as K
import numpy as np
import random

if K.backend() == 'tensorflow':
    import tensorflow as tf


from scipy.stats import chi2
from numpy.linalg import norm

class BatPF(Optimizer):
    def __init__(self,D=7,NP=50,**kwargs):

        super(BatPF,self).__init__(**kwargs)
        with K.name_scope(self.__class__.__name__):
            self.D=D
            self.NP=NP
            self.best=[0]*D
            self.x_obs=[0]*D
            self.z_obs=[0]*D
            self.x_pred=[[0 for i in range (D)] for j in range (NP)]
            self.z_pred=[[0 for i in range (D)] for j in range (NP)]
            self.v=[[0 for i in range (D)] for j in range (NP)]
            self.f=[0]*NP
            self.r=[0.5]*NP
            self.A=[0.5]*NP
            self.process_noise=np.random.normal(0,10,D)
            self.measurement_noise=np.random.normal(0,1,D)
            self.k=0
            self.Fitness=[0]*NP
            self.f_min=0.0
            self.fmin = 0.0
            self.fmax = 2.0
```

```
def get_updates(self,loss,params):

    self.k+=1
    def state_transistion(x,k):
        return ((x/2)+(25*x/(1+x**2))+8*np.math.cos(1.2*k))

    def measurement(x):
        return x*x/20
    #eq 6 and 7
    for j in range (self.D):
        self.x_obs[j]=state_transistion(self.x_obs[j],self.k)+self.process_noise[j]
        self.z_obs[j]=measurement(self.x_obs[j])+self.measurement_noise[j]
    #BAT algo
    #objective function
    def objective(true,predicted,D):
        val=0.0
        for i in range(D):
            val= val+ math.exp(-0.5*(true[i]-predicted[i]))
        #print("val "+ str(val))
        return val
    #x=[x1,...,xd]

    #initialize bat population xi and vis
    #define fi at xi

    #Initialize ri and Ai
    x_new=[[0 for i in range (self.D)] for j in range (self.NP) ]
    t=0

    for i in range(self.NP):
        for j in range(self.D):
            rnd = np.random.uniform(0, 1)
            self.v[i][j] = 0.0
            self.x_pred[i][j] = rnd
            self.z_pred[i][j]=measurement(self.x_pred[i][j])+self.measurement_noise[j]
            #print("z_pred "+str(self.z_pred))
        self.Fitness[i] = objective(self.z_obs,self.z_pred[i],self.D)
        #print ("Fitness"+str(self.Fitness))

    i = 0
    j = 0
    for i in range(self.NP):
        #print(self.Fitness[i])
```

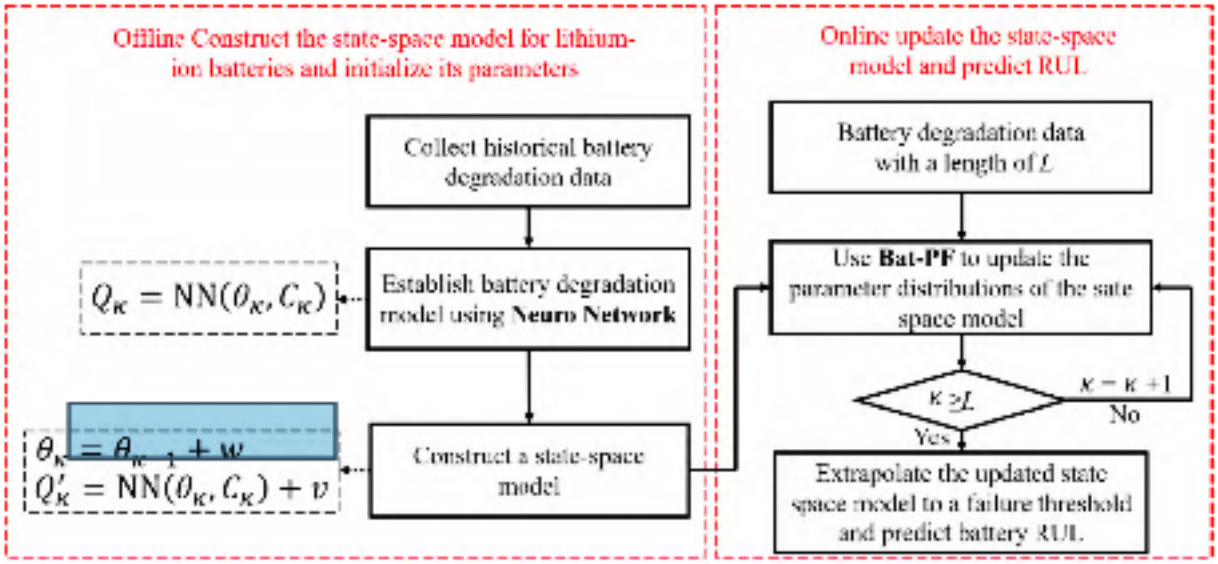
```
    #print(self.Fitness[j])
    if self.Fitness[i] < self.Fitness[j]:
        j = i
for i in range(self.D):
    self.best[i] = self.x_pred[j][i]
self.f_min = self.Fitness[j]

#while(t<max no. iterations)
while(t<1000):
#{ Generate new solutions by adjusting fi,vi,xi using eq 12,13,14
    for i in range(self.NP):
        rnd = np.random.uniform(0, 1)
        self.f[i] = self.fmin + (self.fmax - self.fmin) * rnd
        for j in range(self.D):
            self.v[i][j] = self.v[i][j] + (self.x_pred[i][j] -self.best[j]) * self.f[i]
            self.x_pred[i][j] = self.x_pred[i][j] + self.v[i][j]

        rnd = np.random.random_sample()
        # if(random<ri)
        if(rnd<self.r[i]):
            #{Select a solution among the best solutions
            for j in range(self.D):
                x_new[i][j]=self.best[j] + 0.001 * random.gauss(0, 1)
            #Generate a local solution around the selected best solution
            #xnew = xold + epsilon[-1,1]*Avg loudness
            # }
        self.z_pred[i][j]=measurement(x_new[i][j])+self.measurement_noise[j]
        Fnew = objective(self.z_obs,self.z_pred[i],self.D)
#Generate a solution by flying randomly
        rnd = np.random.random_sample()
#if(random<Ai and I(xi)<I(x*))
        if rnd<self.A[i] and Fnew<self.Fitness[i]:
            for j in range(self.D):
                self.x_pred[i][j] = x_new[i][j]
            self.Fitness[i] = Fnew
#{Accept the new solutions
#Increase ri (r=r0(1-exp(-gamma*t))) and reduce Ai(Ai(t+1)=alpha*Ai(t))}
#Rank the bats and find the current best x*
#}
        if Fnew <= self.f_min:
            for j in range(self.D):
                self.best[j] = x_new[i][j]
            self.f_min = Fnew
```

```
            t+=1
        #End BAT algo


        #get x*
        #eq 10 (chi2pdf)
        i=0
        for p in params:
            new_p=p*K.constant([chi2.pdf(norm(self.z_obs[i]-self.best[i]),2)])
            self.updates.append(K.update(p,new_p))
            i+=1


        return self.updates
        #return self.updates
    def set_weights(self, weights):
        params = self.weights
        # Override set_weights for backward compatibility of Keras 2.2.4 optimizer
        # since it does not include iteration at head of the weight list. Set
        # iteration to 0.
        if len(params) == len(weights):
            self.weights = weights
        super(BatPF, self).set_weights(weights)
    def get_config(self):
        print("cinfig me bhi aya")
        config = {'D': float(K.get_value(self.D)) }
        base_config = super(BatPF, self).get_config()
        return dict(list(base_config.items()) + list(config.items()))




“””Neural Network implementation using keras”””
import pandas as pd

# Import Dataset
dataset = pd.read_csv('Cycle n Capacity.csv')
X=dataset.iloc[:,0:1].values
y=dataset.iloc[:,1].values

# Splitting the dataset into the Training set and Test set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.2,random_state=True)
```

```
# Import keras libraries and packages
import keras
from keras.models import Sequential
from keras.layers import Dense

# Initializing the ANN
regressor = Sequential()

# Adding the input layer and first hidden layer
regressor.add(Dense(output_dim=2 ,init = 'uniform',activation = 'tanh',input_dim = 1))

# Adding the output layer
regressor.add(Dense(output_dim = 1,init = 'uniform',activation = 'linear'))

# Compiling the ANN
regressor.compile(optimizer = BatPF(), loss = 'mean_squared_error', metrics=
['mean_absolute_error'])

# Fitting the ANN in the Training set
regressor.fit(X_train, y_train, batch_size = 1,nb_epoch = 1000)
# Predicting the Test set result
y_pred = regressor.predict(X_test)

weights = regressor.layers[0].get_weights()[0]
biases = regressor.layers[0].get_weights()[1]
weights1 = regressor.layers[1].get_weights()[0]
biases1 = regressor.layers[1].get_weights()[1]

numsum=0
densum=0
mean=0
summ=0
for i in range(0,128):
    summ=summ+y_test[i]

mean=summ/128

for i in range(0,128):
    numsum=numsum+(y_test[i]-y_pred[i])**2
    densum=densum+(y_test[i]-mean)**2

#R^2
```

R2=1-(numsum/densum)
#RMSE
Rmse = math.sqrt(numsum/128)

RUL Prediction:



Predicting the Remaining Useful Life (RUL) of Li-ion batteries :

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 collected 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 Bat-PF method can obtain smaller prediction errors and narrower pdf widths than the standard PF at different prediction times, indicating the Bat-PF improves the prediction performance.

Neural Network using above topology with sgd optimizer:

```
# Import Libraries
import pandas as pd

# Import Dataset
dataset = pd.read_csv('training_data.csv')
X_train=dataset.iloc[:,0:1].values
y_train=dataset.iloc[:,1].values

test_data = pd.read_csv('testing_data.csv')
X_005=test_data.iloc[:,0:1].values
y_005=test_data.iloc[:,1].values

# Splitting the dataset into the Training set and Test set
#from sklearn.model_selection import train_test_split
#X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.2,random_state=True)
```

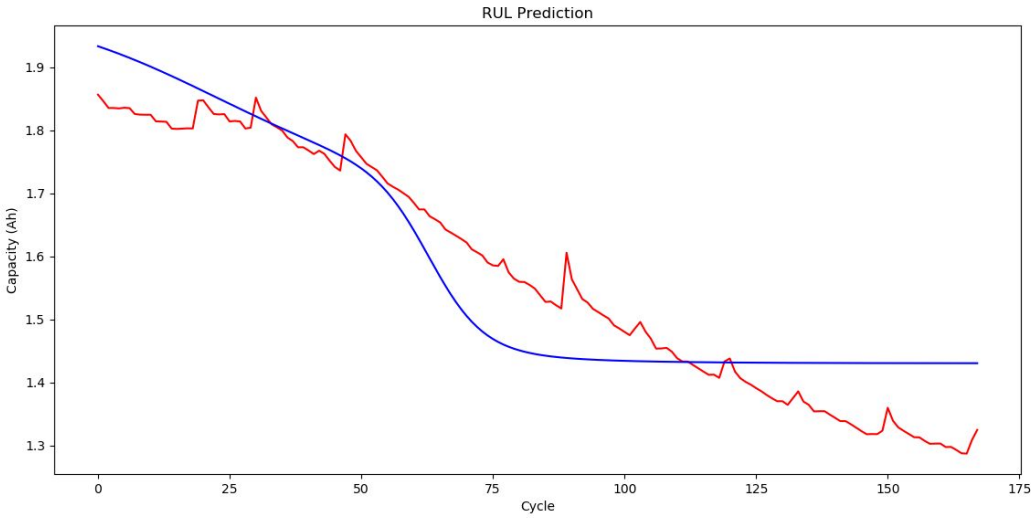
```
# Import keras libraries and packages
import keras
from keras.models import Sequential
from keras.layers import Dense

# Initializing the ANN
regressor = Sequential()

# Adding the input layer and first hidden layer
regressor.add(Dense(output_dim=2 ,init = 'uniform',activation = 'tanh',input_dim = 1))

# Adding the output layer
regressor.add(Dense(output_dim = 1,init = 'uniform',activation = 'linear'))

# Compiling the ANN
regressor.compile(optimizer = 'sgd', loss = 'mean_squared_error', metrics= ['mean_absolute_error'])

# Fitting the ANN in the Training set
regressor.fit(X_train, y_train, batch_size = 1,nb_epoch = 2000)
# Predicting the Test set result
y_pred = regressor.predict(X_005)
y_pred=[]
t=[20]
y_pred=[]

while(t[0]<165 ):
   t[0]+=1
   y_pred.append(regressor.predict(t))

rul=t
weights = regressor.layers[0].get_weights()[0]
biases = regressor.layers[0].get_weights()[1]

numsum=0
densum=0
mean=0
summ=0
for i in range(0,128):
   summ=summ+y_005[i]

mean=summ/128

for i in range(0,128):
```

```
    numsum=numsum+(y_005[i]-y_pred[i])**2
    densum=densum+(y_005[i]-mean)**2

R2=1-(numsum/densum)
X_0050=[]
for x in range(20,165):
    X_0050.append(x)
import matplotlib.pyplot as plt
threshold=1.5
plt.plot(X_005, y_005, color='red')
plt.plot(X_0050, y_pred, color='blue')
plt.axhline(threshold, color='yellow', linestyle='--')
plt.title('RUL Prediction')
plt.xlabel('Cycle')
plt.ylabel('Capacity (Ah)')
plt.show()
```

Graphs and Inferences:

RUL Prediction using SGD optimizer

Details of Neural Network
Input Layer: 1 neuron, Activation function= tanh
Single Hidden layer: 2 neurons, Activation function= linear
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 B0005 battery. 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 not give accurate results.

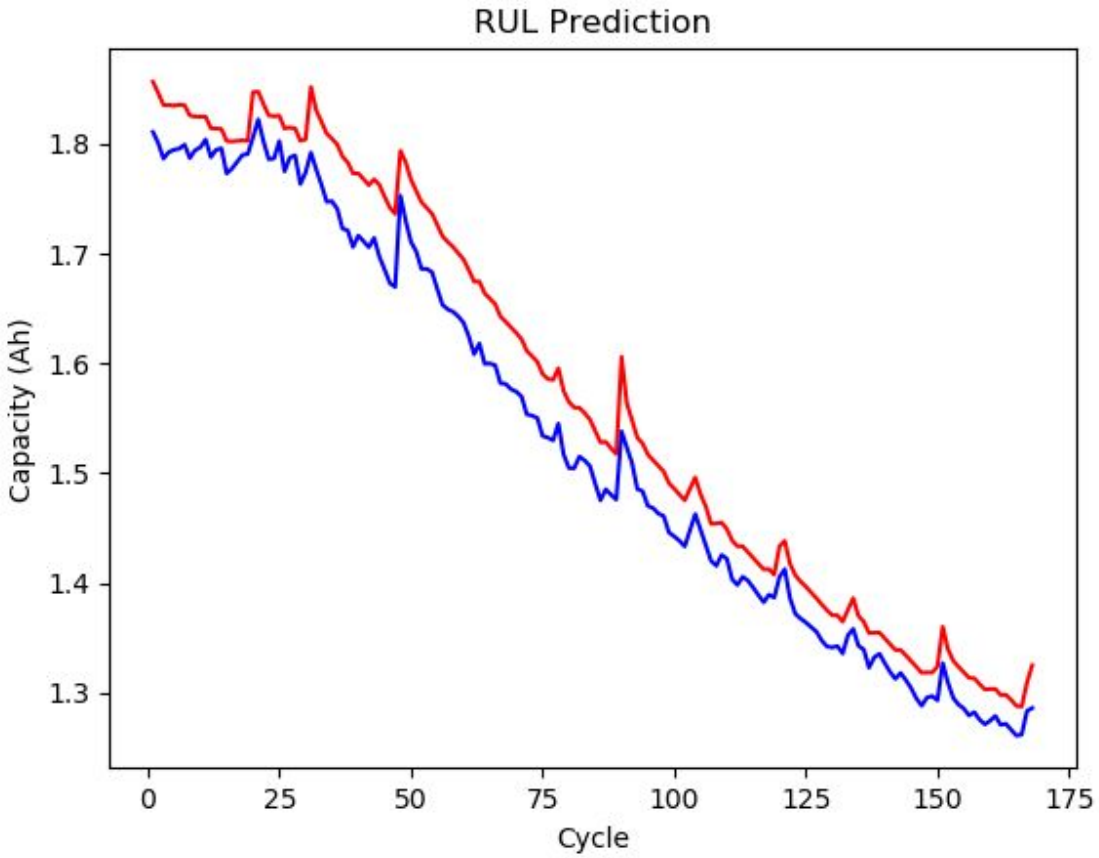
New findings:

Failed Attempts-

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 polynomial regression was applied.
After applying Polynomial regression on Cycle number versus each of the above mentioned features, following values of RMSE and R2 were obtained.

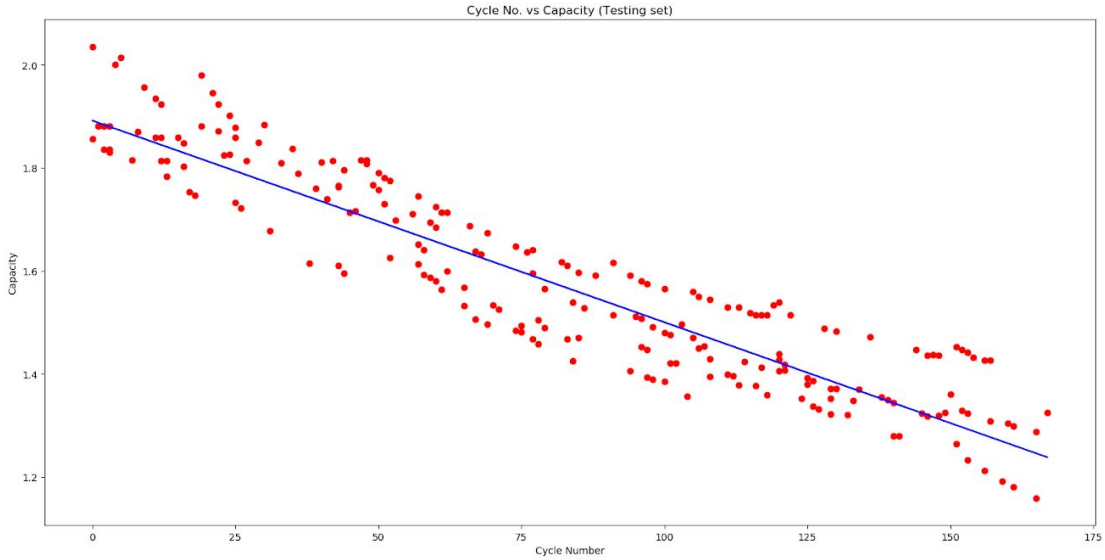
	R2	RMSE
Cycle no. and Time measured	0.91	74
Cycle no. and Voltage measured	0.43	0.298
Cycle no. and Current measured	0.2	0.5
Cycle no. and Temperature measured	0.011	2.22

The value of R2 must tend towards 1.00 and the value of RMSE must tend toward 0 for good accuracy.
Since none of the feature gives a good prediction result (accuracy) it was a failed attempt which proved that none of the feature are polynomially dependent on cycle number
polynomial regression



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.

As per the methodology implemented in the above mentioned research paper, capacity was predicted only using cycle number. The accuracy of predicted capacity was measured using R square(R2) and root mean square error (RMSE) functions.
The values for the obtained errors are as follows:
R2 = 0.859
RMSE = 0.074



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 i.e
R2=0.9766
RMSE=0..0303
This capacity can be used for further calculations.

Interpretations/Comments/Findings:

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.