Remaining Useful Life Prediction of Nasa Turbofan Engine

1. Introduction

Remaining Useful Life prediction is a common problem in the field of Prognostics and many data driven methods are used to predict remaining useful life. Early prediction of RUL can be very helpful to detect failure and it can save huge amount of money to manufacturers and investors.

The PHM 2008 Dataset which contains data related to the Turbofan Engine and its Remaining Useful Life. In this project, a deep learning model is used to predict RUL of Nasa Turbofan Engine by using both Convolution Neural Networks and Long Short-Term Memory Models.

2. Data

The data is obtained from the Public Dataset of Nasa Turbofan Engine Run-To-Failure Simulation. Engine degradation simulation was carried out using C-MAPSS. Different engines were simulated under different combinations of operational conditions and fault modes. Records of several sensors are utilized to characterize fault evolution. The data set was provided by the Prognostics CoE at NASA Ames.

The Dataset contains multivariate time series data. Each dataset contains simulation of many different engines. There are 4 sets of data containing subsets of training and testing data. These subsets contain testing of 100 engines for each set under different operating conditions with sensor readings and its RUL. These engines come with different initial wear and different manufacturing conditions

The dataset contains 26 features containing engine number, operational settings and sensor measurements. These features are used to predict RUL with the help of Deep Learning.

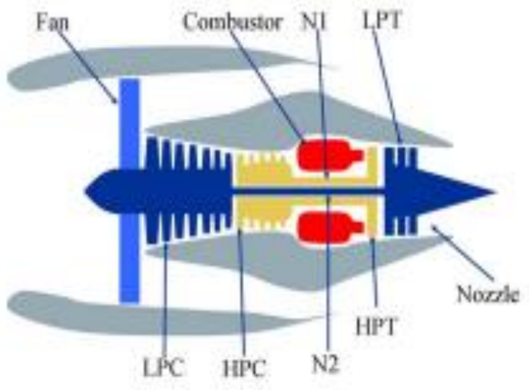


Figure 1. C-MAPSS Simulation of Engine

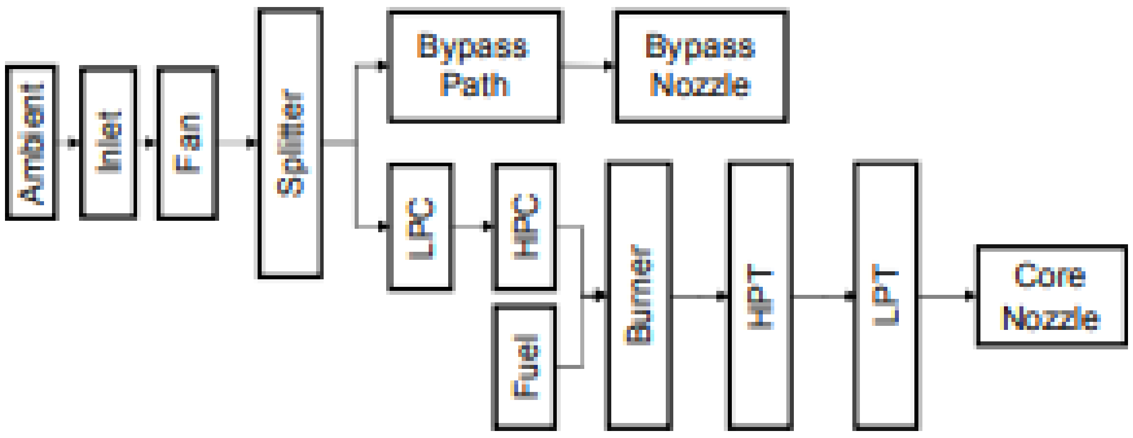


Figure 2. A layout showing various modules and their connections as modelled in simulation.

3. Methodology

A Deep Learning model containing both 1D Convolution Neural Networks and Long Short-Term Memory is used to predict the RUL of the engine. The CNN extracts the spatial features of the data set and the LSTM extracts the temporal features of the dataset. These features are further fed into a CNN and then further into dense layers to predict RUL.

The C-MAPSS dataset is cleaned and divided into CNN input and LSTM input. The CNN Layer contains three 1D Conv layers, three Max Pool Layers with Batch Normalization. The CNN layers have 128, 64 and 32 filters, kernel size of 3 and activation function as 'RELU'. The pooling layer has a pool size of 2, stride as 2 and padding as 'same'. The first two CNN layers are batch normalized. The LSTM layer has 3 LSTM units with 128, 64 and 32 respectively. Both the outputs from CNN and LSTM layers are concatenated and fed into a 1-D CNN layer with 256 filters and kernel size of 1. It is fed to a Max Pool layer with pool size as 2, stride as 2 and padding set to 'same'. The output is then flattened and fed into two dense layers consisting of 128 and 32 units with intermediate dropout value of 0.3 and activation function set to 'RELU'. Finally, a single unit dense layer gives us the RUL.

The model is compiled with Adam Optimizer and a Custom Loss function i.e., Root Mean Square Error. Root Mean Square Error is also used as the metric which measures the accuracy of the model.

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_{prediction} - y_{true})^{\,2}}$$

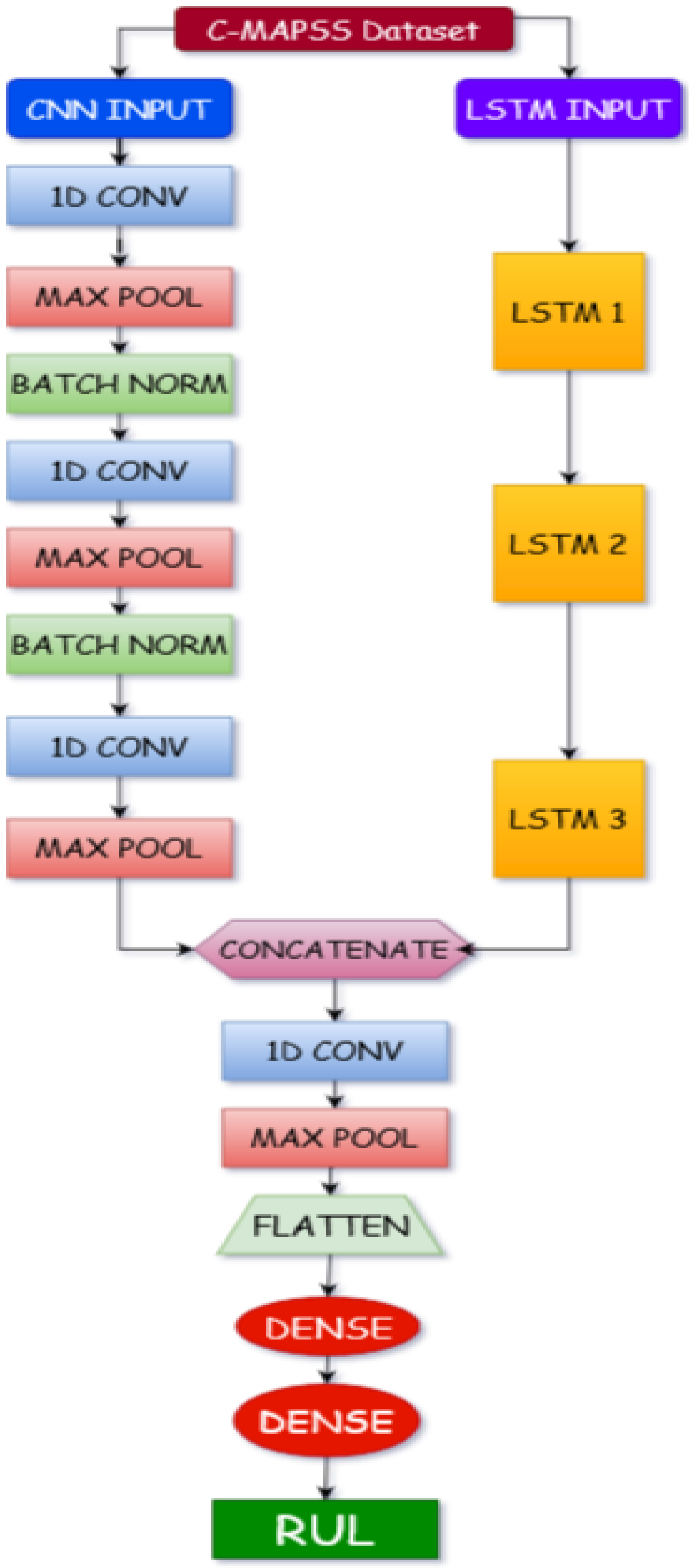
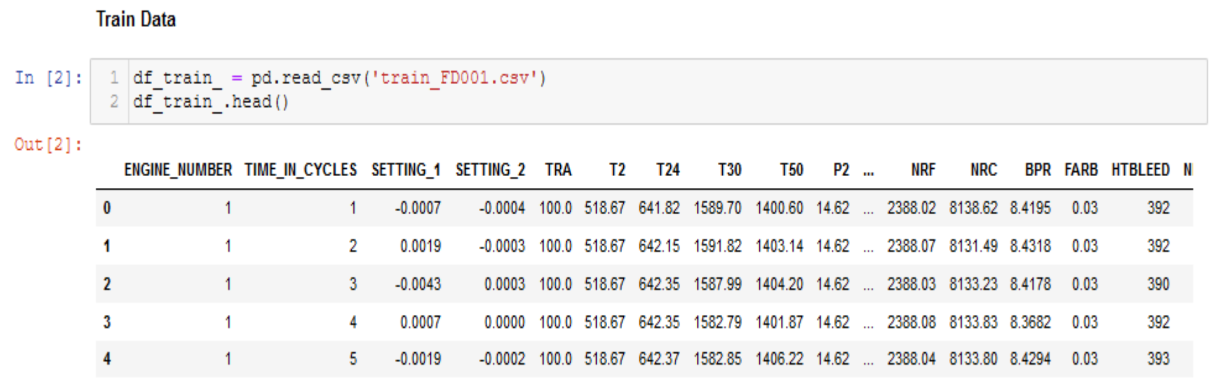


Figure 3. Architecture of the Neural Network

4. Analysis

4.1 Data Pre-processing

‘FD001’ dataset is used as the training and testing data. The train dataset and test data are read. The test dataset does not contain the ‘RUL’ but another file with ‘RUL’ is given. With a particular engine number in test data set, the RUL after running the series of real time simulation is given in the ‘RUL File’. It is this value that we must achieve to obtain with our model after training.



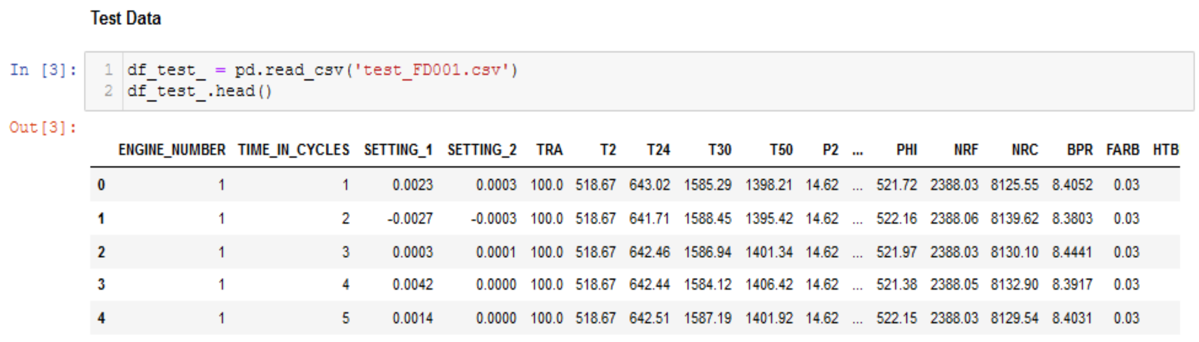
Train Data

In [2]:
```
df_train_ = pd.read_csv('train_FD001.csv')
df_train_.head()
```

Out[2]:

	ENGINE_NUMBER	TIME_IN_CYCLES	SETTING_1	SETTING_2	TRA	T2	T24	T30	T50	P2	...	NRF	NRC	BPR	FARB	HTBLEED	N
0	1	1	-0.0007	-0.0004	100.0	518.67	641.82	1589.70	1400.60	14.62	...	2388.02	8138.62	8.4195	0.03	392	
1	1	2	0.0019	-0.0003	100.0	518.67	642.15	1591.82	1403.14	14.62	...	2388.07	8131.49	8.4318	0.03	392	
2	1	3	-0.0043	0.0003	100.0	518.67	642.35	1587.99	1404.20	14.62	...	2388.03	8133.23	8.4178	0.03	390	
3	1	4	0.0007	0.0000	100.0	518.67	642.35	1582.79	1401.87	14.62	...	2388.08	8133.83	8.3682	0.03	392	
4	1	5	-0.0019	-0.0002	100.0	518.67	642.37	1582.85	1406.22	14.62	...	2388.04	8133.80	8.4294	0.03	393	

Figure 4. Train Data of FD001.



Test Data

In [3]:
```
df_test_ = pd.read_csv('test_FD001.csv')
df_test_.head()
```

Out[3]:

	ENGINE_NUMBER	TIME_IN_CYCLES	SETTING_1	SETTING_2	TRA	T2	T24	T30	T50	P2	...	PHI	NRF	NRC	BPR	FARB	HTB
0	1	1	0.0023	0.0003	100.0	518.67	643.02	1585.29	1398.21	14.62	...	521.72	2388.03	8125.55	8.4052	0.03	
1	1	2	-0.0027	-0.0003	100.0	518.67	641.71	1588.45	1395.42	14.62	...	522.16	2388.06	8139.62	8.3803	0.03	
2	1	3	0.0003	0.0001	100.0	518.67	642.46	1586.94	1401.34	14.62	...	521.97	2388.03	8130.10	8.4441	0.03	
3	1	4	0.0042	0.0000	100.0	518.67	642.44	1584.12	1406.42	14.62	...	521.38	2388.05	8132.90	8.3917	0.03	
4	1	5	0.0014	0.0000	100.0	518.67	642.51	1587.19	1401.92	14.62	...	522.15	2388.03	8129.54	8.4031	0.03	

Figure 5. Test Data of FD001.

4.1.1. Feature Extraction:

In order to predict the RUL of the Engine, we need useful feature which will influence the value of the RUL. So, we plot a correlation matrix to see how each feature is correlated with the other feature. A correlation of 1 means the features are positively correlated, a correlation value of -1 means the features are negatively correlated and a correlation value of 0 means the features are not correlated.

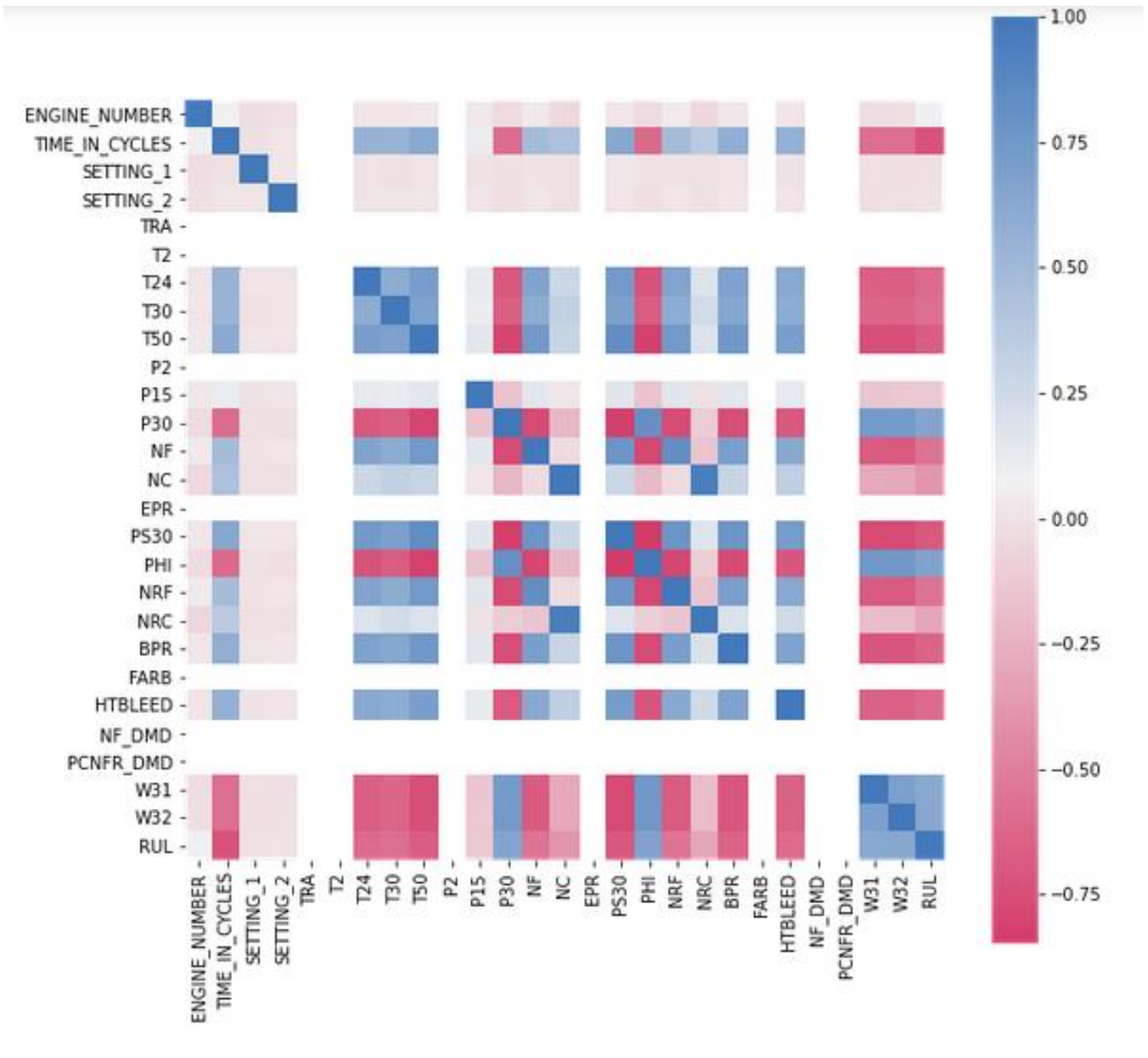
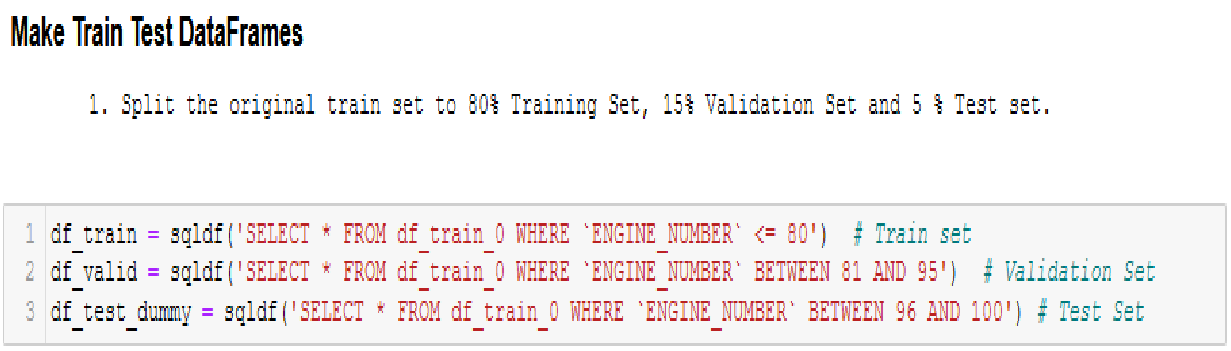


Figure 6.Correlation Matrix plotted using Heatmap of Seaborn.

Features like 'TRA', 'T2', 'P2', 'EPR', 'FARB', 'NF_DMD', 'PCNFR_DMD' have ‘Nan’ values, these features are dropped. If we look at the data, these features do not change with time and are constants. Similarly, ‘ENGINE_NUMBER’, ‘SETTING_1’ and ‘SETTING_2’ have correlation close to zero. Hence these features will be dropped. Thus, we have 16 features and 1 target i.e. RUL.

4.1.3 Train-Test-Valid Split:

The training dataset is further split into 80% training, 15% validation and 5% test data. Splitting is done based on Engine Numbers. Engine number having number below 80 are training data. Engine number from 81 to 95 are validation data. Engine number greater than 96 is testing data.



Make Train Test DataFrames

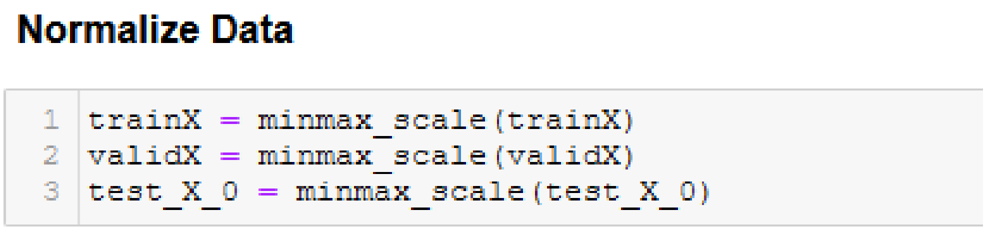
1. Split the original train set to 80% Training Set, 15% Validation Set and 5 % Test set.

```
df_train = sqldf('SELECT * FROM df_train_0 WHERE `ENGINE_NUMBER` <= 80')  # Train set
df_valid = sqldf('SELECT * FROM df_train_0 WHERE `ENGINE_NUMBER` BETWEEN 81 AND 95')  # Validation Set
df_test_dummy = sqldf('SELECT * FROM df_train_0 WHERE `ENGINE_NUMBER` BETWEEN 96 AND 100') # Test Set
```

Figure 7. Train-Test-Valid Split.

4.1.3 Normalization:

Normalizing is done to bring all the features to a similar scale so that some features are not biased while training. The data set is normalized using sklearn's preprocessing package(minmax_scale).



Normalize Data

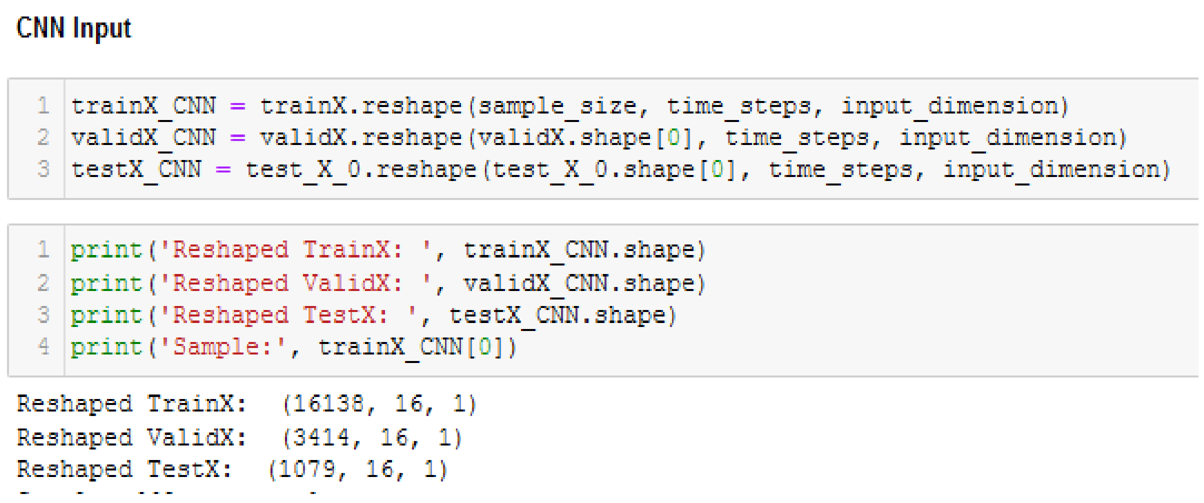
```
trainX = minmax_scale(trainX)
validX = minmax_scale(validX)
test_X_0 = minmax_scale(test_X_0)
```

Figure 8. Normalize Data.

4.2 Data Analysis:

4.2.1 Create Inputs for CNN and LSTM Layers

Separate inputs have to be made in order to feed the CNN and LSTM layers. CNN expects a 3-D Tensor with shape as "[batch_size, time_steps, input_dimension]" and LSTM expects a 3-D Tensor with shape as "[batch_size, time_steps, input_dimension]" .



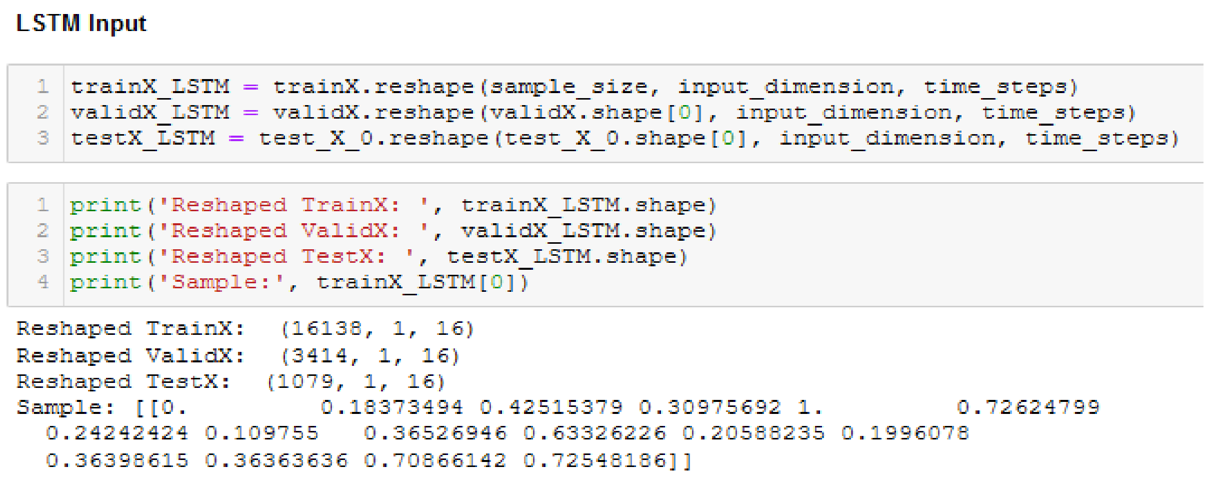
CNN Input

```
trainX_CNN = trainX.reshape(sample_size, time_steps, input_dimension)
validX_CNN = validX.reshape(validX.shape[0], time_steps, input_dimension)
testX_CNN = test_X_0.reshape(test_X_0.shape[0], time_steps, input_dimension)
```

```
print('Reshaped TrainX: ', trainX_CNN.shape)
print('Reshaped ValidX: ', validX_CNN.shape)
print('Reshaped TestX: ', testX_CNN.shape)
print('Sample:', trainX_CNN[0])
```

```
Reshaped TrainX:  (16138, 16, 1)
Reshaped ValidX:  (3414, 16, 1)
Reshaped TestX:  (1079, 16, 1)
```

Figure 8. CNN Input



LSTM Input

```
trainX_LSTM = trainX.reshape(sample_size, input_dimension, time_steps)
validX_LSTM = validX.reshape(validX.shape[0], input_dimension, time_steps)
testX_LSTM = test_X_0.reshape(test_X_0.shape[0], input_dimension, time_steps)
```

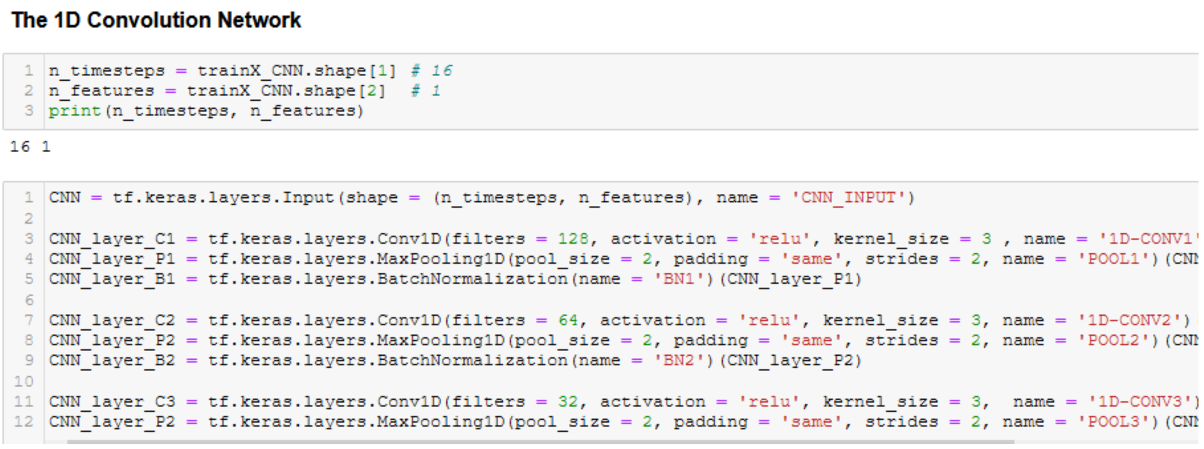
```
print('Reshaped TrainX: ', trainX_LSTM.shape)
print('Reshaped ValidX: ', validX_LSTM.shape)
print('Reshaped TestX: ', testX_LSTM.shape)
print('Sample:', trainX_LSTM[0])
```

```
Reshaped TrainX:  (16138, 1, 16)
Reshaped ValidX:  (3414, 1, 16)
Reshaped TestX:  (1079, 1, 16)
Sample: [[0.         0.18373494 0.42515379 0.30975692 1.         0.72624799
  0.24242424 0.109755   0.36526946 0.63326226 0.20588235 0.1996078
  0.36398615 0.36363636 0.70866142 0.72548186]]
```

Figure 9. LSTM Input.

4.2.2 Define the Neural Network

The neural network is defined by using the TensorFlow's Keras package. The neural network is to be designed as per the architecture mentioned in Section 3.



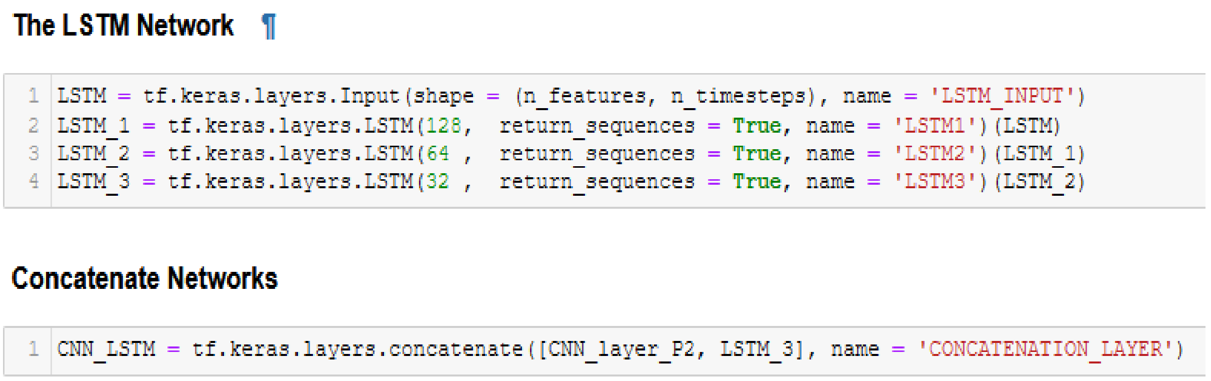
The 1D Convolution Network

```
n_timesteps = trainX_CNN.shape[1] # 16
n_features = trainX_CNN.shape[2]  # 1
print(n_timesteps, n_features)
```

```
16 1
```

```
CNN = tf.keras.layers.Input(shape = (n_timesteps, n_features), name = 'CNN_INPUT')

CNN_layer_C1 = tf.keras.layers.Conv1D(filters = 128, activation = 'relu', kernel_size = 3 , name = '1D-CONV1'
CNN_layer_P1 = tf.keras.layers.MaxPooling1D(pool_size = 2, padding = 'same', strides = 2, name = 'POOL1')(CN
CNN_layer_B1 = tf.keras.layers.BatchNormalization(name = 'BN1')(CNN_layer_P1)

CNN_layer_C2 = tf.keras.layers.Conv1D(filters = 64, activation = 'relu', kernel_size = 3, name = '1D-CONV2')
CNN_layer_P2 = tf.keras.layers.MaxPooling1D(pool_size = 2, padding = 'same', strides = 2, name = 'POOL2')(CN
CNN_layer_B2 = tf.keras.layers.BatchNormalization(name = 'BN2')(CNN_layer_P2)

CNN_layer_C3 = tf.keras.layers.Conv1D(filters = 32, activation = 'relu', kernel_size = 3,  name = '1D-CONV3')
CNN_layer_P2 = tf.keras.layers.MaxPooling1D(pool_size = 2, padding = 'same', strides = 2, name = 'POOL3')(CN
```

Figure 10. CNN Network.



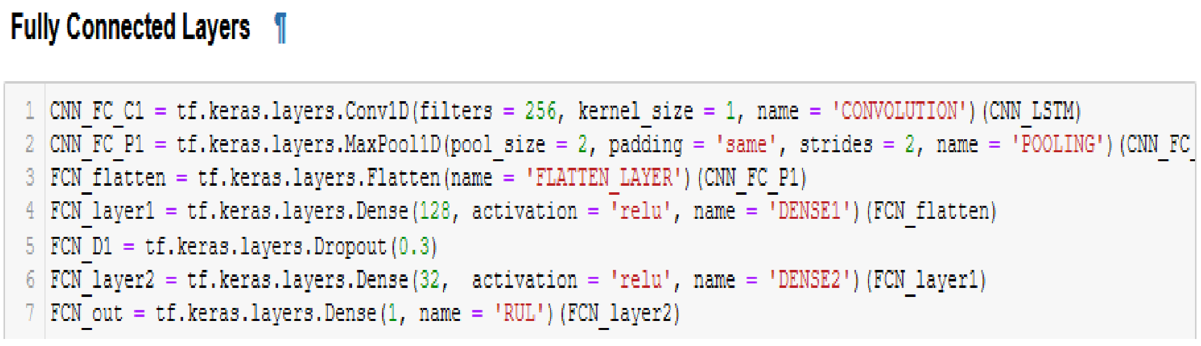
The LSTM Network ¶

```
LSTM = tf.keras.layers.Input(shape = (n_features, n_timesteps), name = 'LSTM_INPUT')
LSTM_1 = tf.keras.layers.LSTM(128,  return_sequences = True, name = 'LSTM1')(LSTM)
LSTM_2 = tf.keras.layers.LSTM(64 ,  return_sequences = True, name = 'LSTM2')(LSTM_1)
LSTM_3 = tf.keras.layers.LSTM(32 ,  return_sequences = True, name = 'LSTM3')(LSTM_2)
```

Concatenate Networks

```
CNN_LSTM = tf.keras.layers.concatenate([CNN_layer_P2, LSTM_3], name = 'CONCATENATION_LAYER')
```

Figure 11. LSTM and Concatenation Layers.



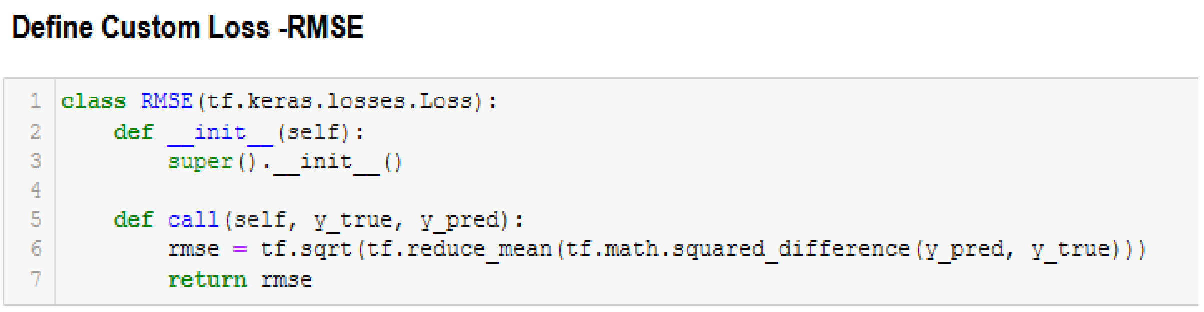
Fully Connected Layers ¶

```
CNN_FC_C1 = tf.keras.layers.Conv1D(filters = 256, kernel_size = 1, name = 'CONVOLUTION')(CNN_LSTM)
CNN_FC_P1 = tf.keras.layers.MaxPool1D(pool_size = 2, padding = 'same', strides = 2, name = 'POOLING')(CNN_FC_
FCN_flatten = tf.keras.layers.Flatten(name = 'FLATTEN_LAYER')(CNN_FC_P1)
FCN_layer1 = tf.keras.layers.Dense(128, activation = 'relu', name = 'DENSE1')(FCN_flatten)
FCN_D1 = tf.keras.layers.Dropout(0.3)
FCN_layer2 = tf.keras.layers.Dense(32,  activation = 'relu', name = 'DENSE2')(FCN_layer1)
FCN_out = tf.keras.layers.Dense(1, name = 'RUL')(FCN_layer2)
```

Figure 12. Fully Connected Layers.

4.2.3 Define Custom Loss:

Keras does not have a Root Mean Square Error loss function. So, we can make a custom loss function by inheriting the Loss class of keras.



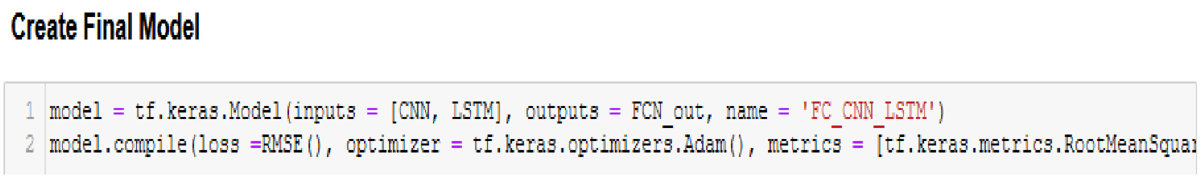
Define Custom Loss -RMSE

```python
class RMSE(tf.keras.losses.Loss):
    def __init__(self):
        super().__init__()

    def call(self, y_true, y_pred):
        rmse = tf.sqrt(tf.reduce_mean(tf.math.squared_difference(y_pred, y_true)))
        return rmse
```

Figure 13. Implementation of Custom Loss Function.

4.2.4 Compile Model:

The model uses Adam optimizer and Loss function defined above. The Root mean Squared error is used as the metrics to measure accuracy.



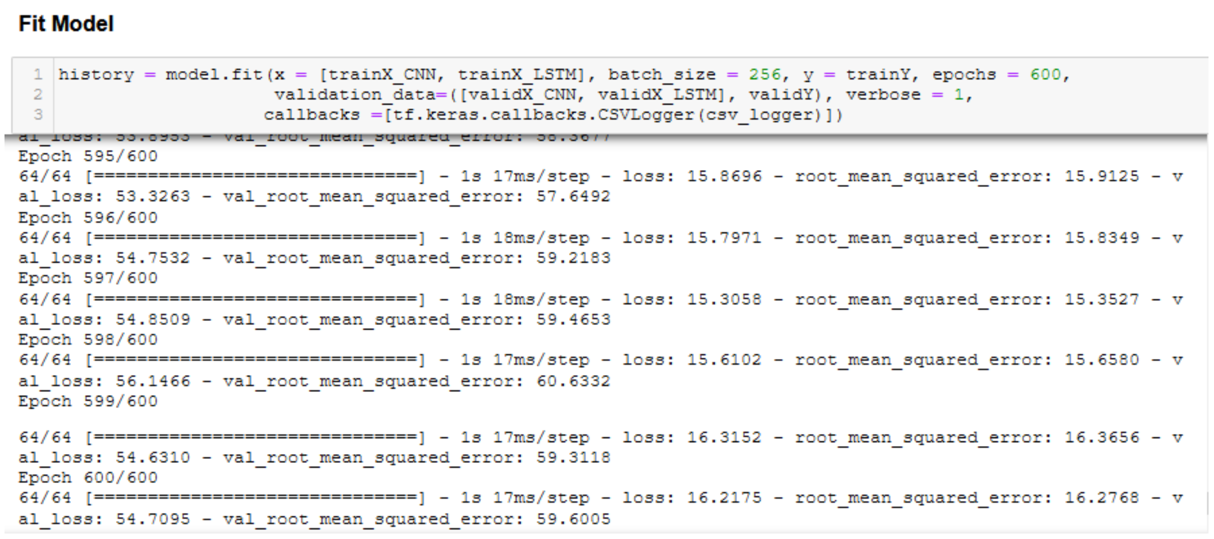
Create Final Model

```python
model = tf.keras.Model(inputs = [CNN, LSTM], outputs = FCN_out, name = 'FC_CNN_LSTM')
model.compile(loss =RMSE(), optimizer = tf.keras.optimizers.Adam(), metrics = [tf.keras.metrics.RootMeanSquar
```

Figure 14. Compiling Model.

4.2.5 Train Model

The model is trained for 600 epochs. A call-back function to record the losses and accuracy is passes. Both train and validation set is passed to the fit method.



Fit Model

```python
history = model.fit(x = [trainX_CNN, trainX_LSTM], batch_size = 256, y = trainY, epochs = 600,
                    validation_data=([validX_CNN, validX_LSTM], validY), verbose = 1,
                   callbacks =[tf.keras.callbacks.CSVLogger(csv_logger)])
```

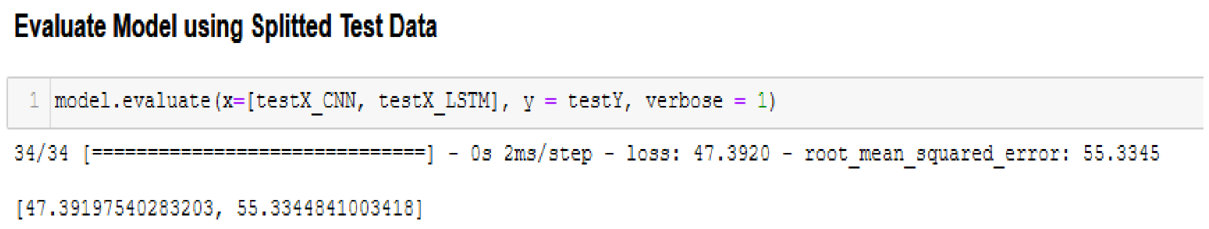
```
Epoch 595/600
64/64 [==============================] - 1s 17ms/step - loss: 15.8696 - root_mean_squared_error: 15.9125 - v
al_loss: 53.3263 - val_root_mean_squared_error: 57.6492
Epoch 596/600
64/64 [==============================] - 1s 18ms/step - loss: 15.7971 - root_mean_squared_error: 15.8349 - v
al_loss: 54.7532 - val_root_mean_squared_error: 59.2183
Epoch 597/600
64/64 [==============================] - 1s 18ms/step - loss: 15.3058 - root_mean_squared_error: 15.3527 - v
al_loss: 54.8509 - val_root_mean_squared_error: 59.4653
Epoch 598/600
64/64 [==============================] - 1s 17ms/step - loss: 15.6102 - root_mean_squared_error: 15.6580 - v
al_loss: 56.1466 - val_root_mean_squared_error: 60.6332
Epoch 599/600

64/64 [==============================] - 1s 17ms/step - loss: 16.3152 - root_mean_squared_error: 16.3656 - v
al_loss: 54.6310 - val_root_mean_squared_error: 59.3118
Epoch 600/600
64/64 [==============================] - 1s 17ms/step - loss: 16.2175 - root_mean_squared_error: 16.2768 - v
al_loss: 54.7095 - val_root_mean_squared_error: 59.6005
```

Figure 15. Training Process

5. Results

5.1 Evaluate Model

The test set is passed to check the accuracy of the model on the test data. Both losses and accuracy were found to be 47.3 and 55.33.



Evaluate Model using Splitted Test Data

```
model.evaluate(x=[testX_CNN, testX_LSTM], y = testY, verbose = 1)
```

```
34/34 [==============================] - 0s 2ms/step - loss: 47.3920 - root_mean_squared_error: 55.3345

[47.39197540283203, 55.3344841003418]
```

Figure 16. Evaluate Model

5.2 Predict on split-test set:

The split test set is passed to the model to get the predicted RUL values. It is also plotted against the True RUL.

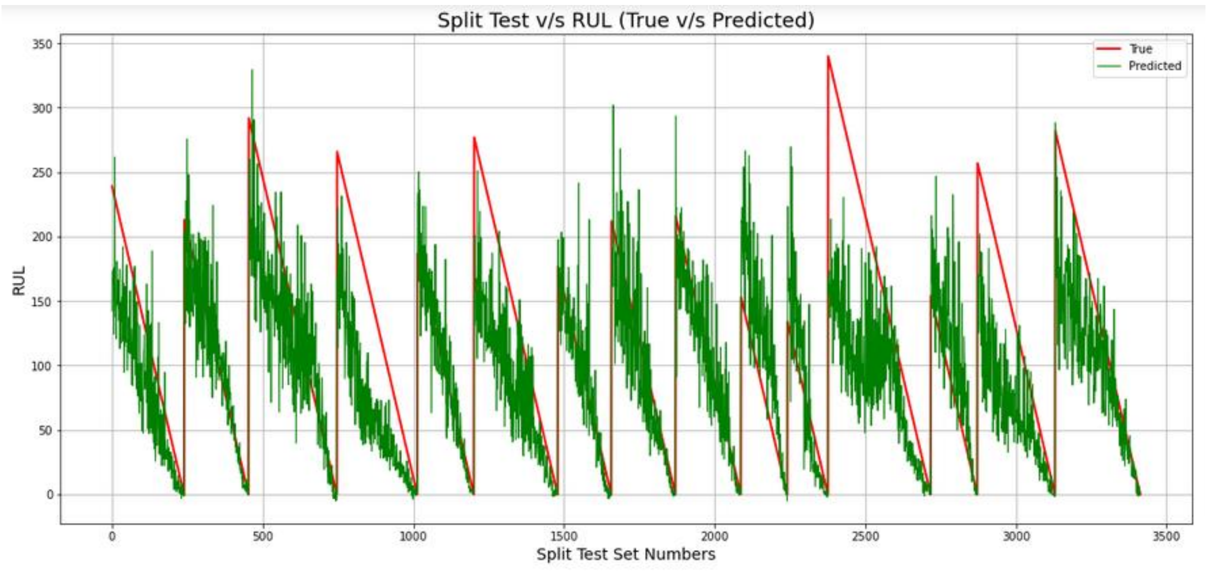


Figure 17. True RUL v/s Predicted RUL

5.3 Plot Losses and Metrics:

The Losses and metrics are visualised to see how they have changed over epochs.

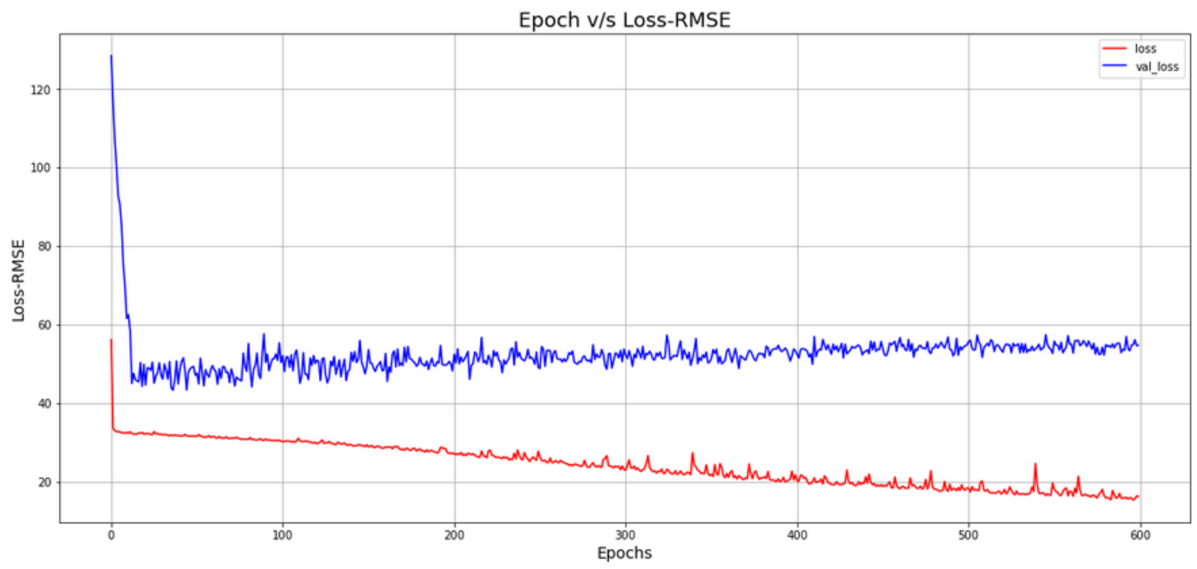


Figure 18. Epochs v/s Loss-RMSE

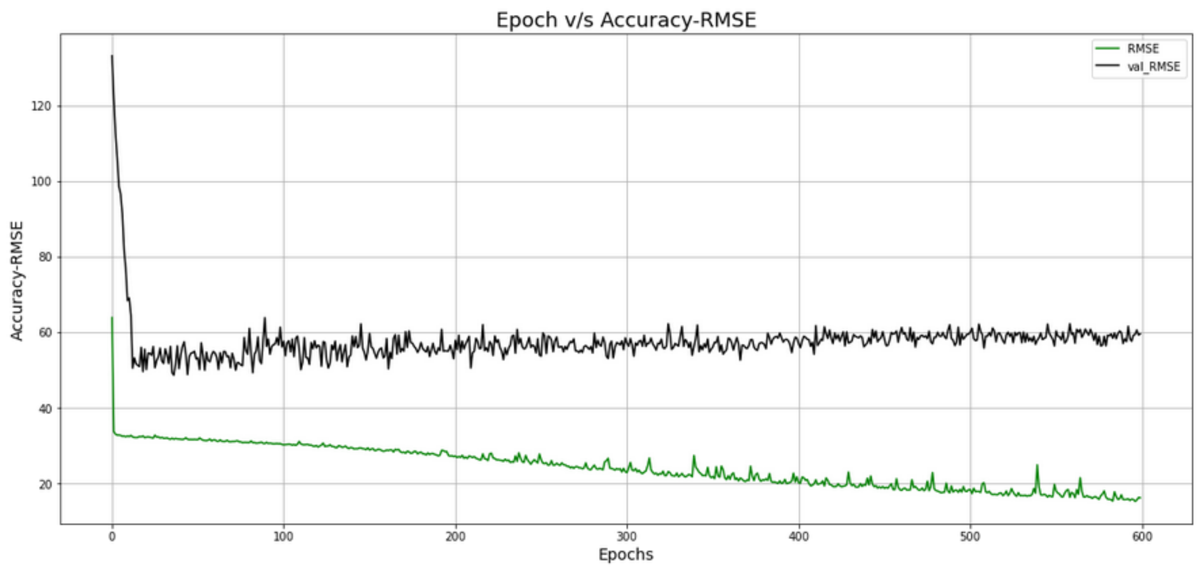


Figure 19. Epochs v/s Accuracy-RMSE

5.4 Predict on Test Set:

The test set which is passed to the model to predict the RUL. An Engine number is selected and passed to the model to predict the RUL and it is compared with the true RUL given in the dataset.

```
Engine Number:  5
RMSE:  43.88896
True RUL:  91
Predicted RUL:  134.88896
```

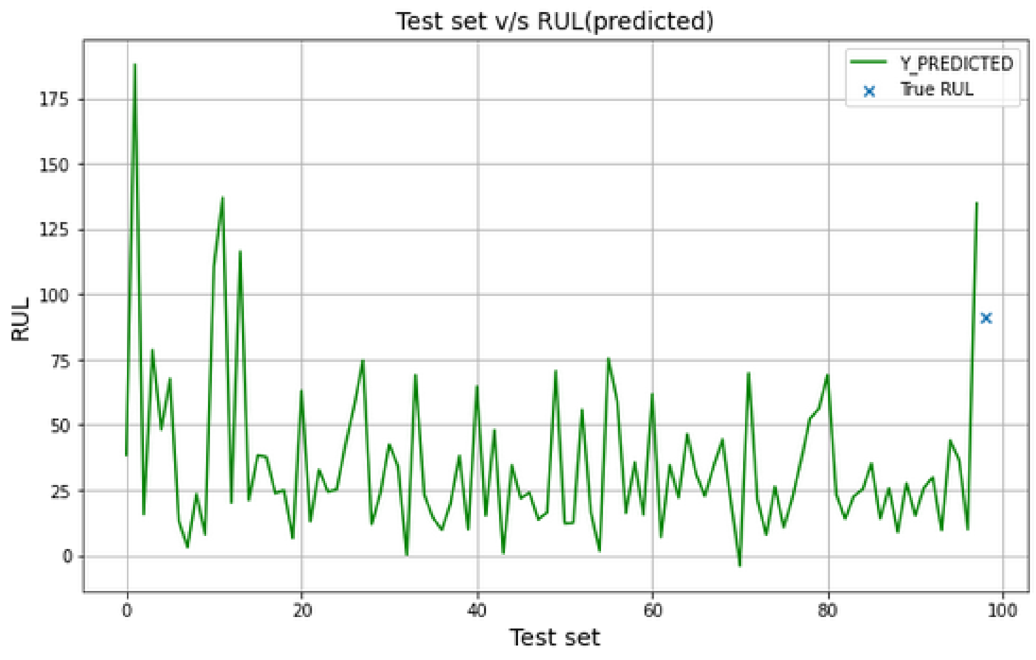


Figure 20. Predicting on the Test Set

7. References:

The following are the references used in this project

1. Reference Paper:

A Remaining Useful Life Prognosis of Turbofan Engine Using Temporal and Spatial Feature Fusion Cheng Peng, Yufeng Chen, Qing Chen, Zhaohui Tang, Lingling Li and Weihua Gui.

2. Reference Paper:

Damage Propagation Modelling for Aircraft Engine Run-to-Failure Simulation Abhinav Saxena, Member IEEE, Kai Goebel, Don Simon, Member, IEEE, Neil Eklund, Member IEEE

3. Dataset: https://www.kaggle.com/behrad3d/nasa-cmaps