Capstone Project

Udacity Machine Learning Engineer Nanodegree

Vikram Haravu (v.haravu@gmail.com)

Feb 2018

Contents

I. Definition

Project Overview

Rechargeable batteries are a very important component of several daily-use gadgets and EV automobiles too. Demand for Lithium batteries has grown tremendously in the last decade and is expected to continue growing in the foreseeable future. As a result, electronics suppliers and OEMs invest hugely on Battery Management Systems (BMS). Central to the BMS is the function of estimating the State of Charge (SOC) of a given battery. This battery parameter called SoC is directly related to the energy stored inside the battery at any given point of time.

In this project, I developed a Machine Learning model to predict the SoC of a rechargeable battery. It is a regression model since SoC lies between 0.0 (empty battery) and 1.0 (fully charged).

Dedicated and expensive hardware is designed into mobile device ICs to perform the SoC estimation function. I explore an ML method to solve this problem. This project was inspired by this IEEE publication.

Problem Statement

The following strategy is followed:

1. Generate data using hardware (RTL) simulations. This is a large task by itself. Briefly, a SystemVerilog testbench is built around SoC estimation hardware. The testbench provides stimulus and observes the functioning of the hardware and writes out data files in CSV format.
2. Normalize and Scale the data – since the features/target are different scales of magnitude and units, it is good practice to normalize and standardize them for optimal algorithm performance.
3. Develop a simple benchmark model, to further assess the problem.
4. Develop a model that will better model non-linearities, as measured by metric scores.

For the project, I will address one phase of the BMS/SoC problem – namely, making accurate prediction of the battery's "*first SoC*" – when a device is powered-on via a battery for the first time. The continuous estimation of SoC during normal operation of a device is critically dependent on accurate estimation of the first SoC. That is because, in the event the first SoC is inaccurate, continuous SoC estimation cannot converge to the correct SoC.

Metrics

Two metrics will be used in this project – R^2 Coefficient of Determination, and MSE Mean Squared Error.

R^2 is the square of the correlation (r) between predicted scores and actual scores. It ranges from 0.0 to 1.0. A model with an R^2=0.0 (worst score) is a model that always predicts the mean of the target variable. On the other hand, a model with an R^2=1.0 (best score) is one that perfectly predicts the target, in other words y_pred = y_test.

MSE gives an indication of how close the predicted values are to the real values. MSE is somewhat subjective (data range dependent) in the sense that it doesn't lie in a fixed range like R^2; a smaller MSE is obviously better. Given the mean of the target will be close to 0.0, and the range is approximately -1.0 to +1.0, an MSE of close to 0.0 (i.e., farthest away from 1.0) is a good value to achieve.

Keras models and optimizers will fundamentally attempt to minimize a loss function. MSE is chosen as that loss function in this project.

II. Analysis

Data Exploration

There are only a few proxies/features that are directly measurable and can be thought of as good indicators of the energy stored inside a rechargeable battery. These features are the voltage measured at the battery terminals (vbatt, volts), the current drawn from (or supplied to) a battery (ibatt, amps) and the operating temperature (tempr, Celsius).

Fig. 1: These three features and the target variable (socMon) are generated using hardware simulations.

vbatt	ibatt	tempr	socMon
389	25	005e	93
391	12	78	98
374	55	36	0
376	45	003f	0
399	11	67	ad
383	14	98	80
375	39	94	5f
407	34	007e	c9
415	19	63	d9
419	47	99	e8
406	69	45	80
412	34	47	d0
406	14	58	c1
385	57	51	76
382	41	79	80
394	66	003a	0
412	17	36	c7

vbatt	ibatt	tempr	socMon
3.89	0.25	23.50	0.576471
3.91	0.12	30.00	0.596078
3.74	0.55	13.50	0.000000
3.76	0.45	15.75	0.000000
3.99	0.11	25.75	0.678431
3.83	0.14	38.00	0.501961
3.75	0.39	37.00	0.372549
4.07	0.34	31.50	0.788235
4.15	0.19	24.75	0.850980
4.19	0.47	38.25	0.909804

The data collected from simulations is shown above. The raw data (CSV file format) is on the left. The raw data needs some minor post processing (such as hex to decimal conversion, divide by 255, etc.) to result in actual v/i/tempr/SoC values shown on the right. I chose to not do this raw to actual conversions in the SystemVerilog simulation domain, because it is much simpler to do so with Python.

Fig. 2: Here are statistical characteristics of the features and target:

	vbatt	ibatt	tempr	socMon
count	1005.000000	1005.000000	1005.000000	1005.000000
mean	3.928169	0.390995	25.175124	0.538051
std	0.163986	0.174999	7.450521	0.272412
min	3.650000	0.100000	13.000000	0.000000
25%	3.780000	0.250000	18.500000	0.333333
50%	3.930000	0.380000	24.750000	0.607843
75%	4.070000	0.540000	31.500000	0.768627
max	4.200000	0.700000	38.750000	0.933333

A note on outliers: The simulations to collect data were run under a controlled set of input stimulus. For instance, the hardware module under simulation is spec'd to operate to a max temperature of 39C. The specific battery being acted upon (for SoC est) is rated between 3.6V to 4.2V. These specifications were adhered to when developing simulation stimulus. So, I chose not to remove outliers.

Exploratory Visualization

Fig. 3: In the figure below, I have plotted each feature on individual x-axes with the target socMon on y-axis. One can see the strong or subtle influence of each feature on the target. It also shows the direct or inverse relation between each feature and target.

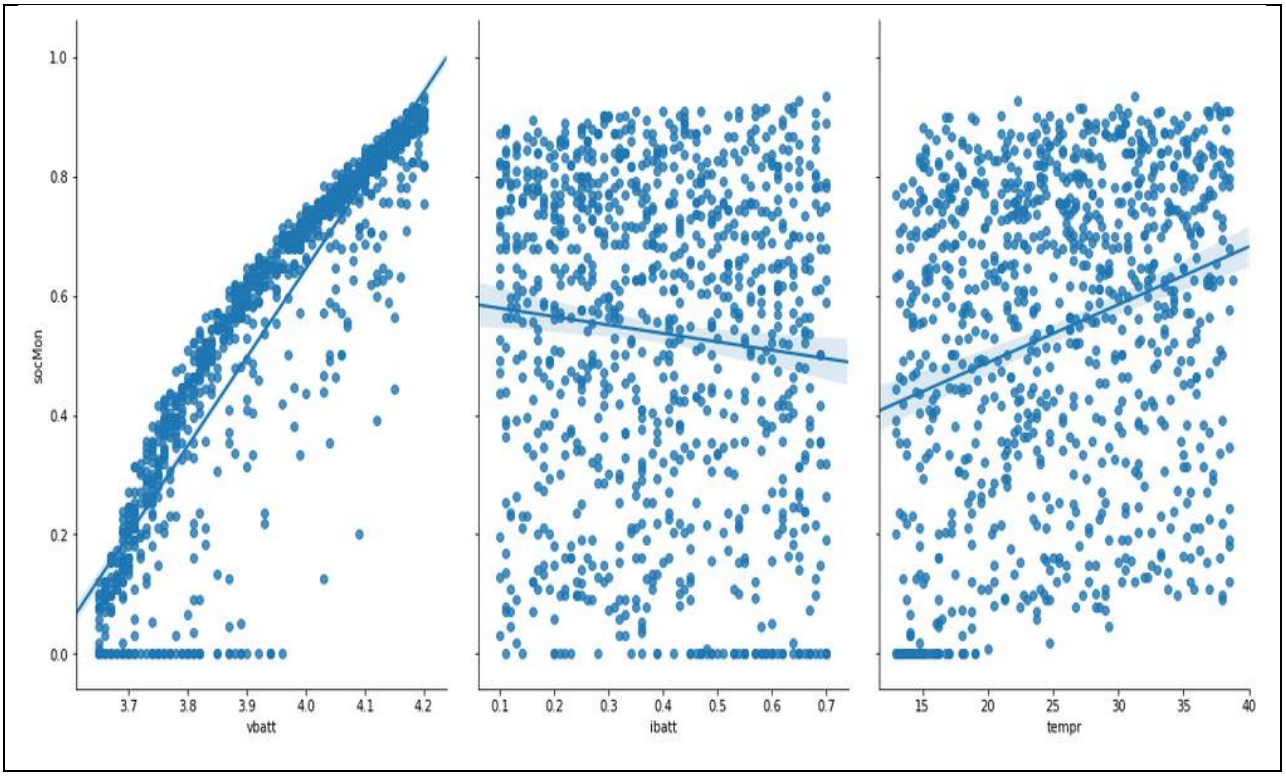
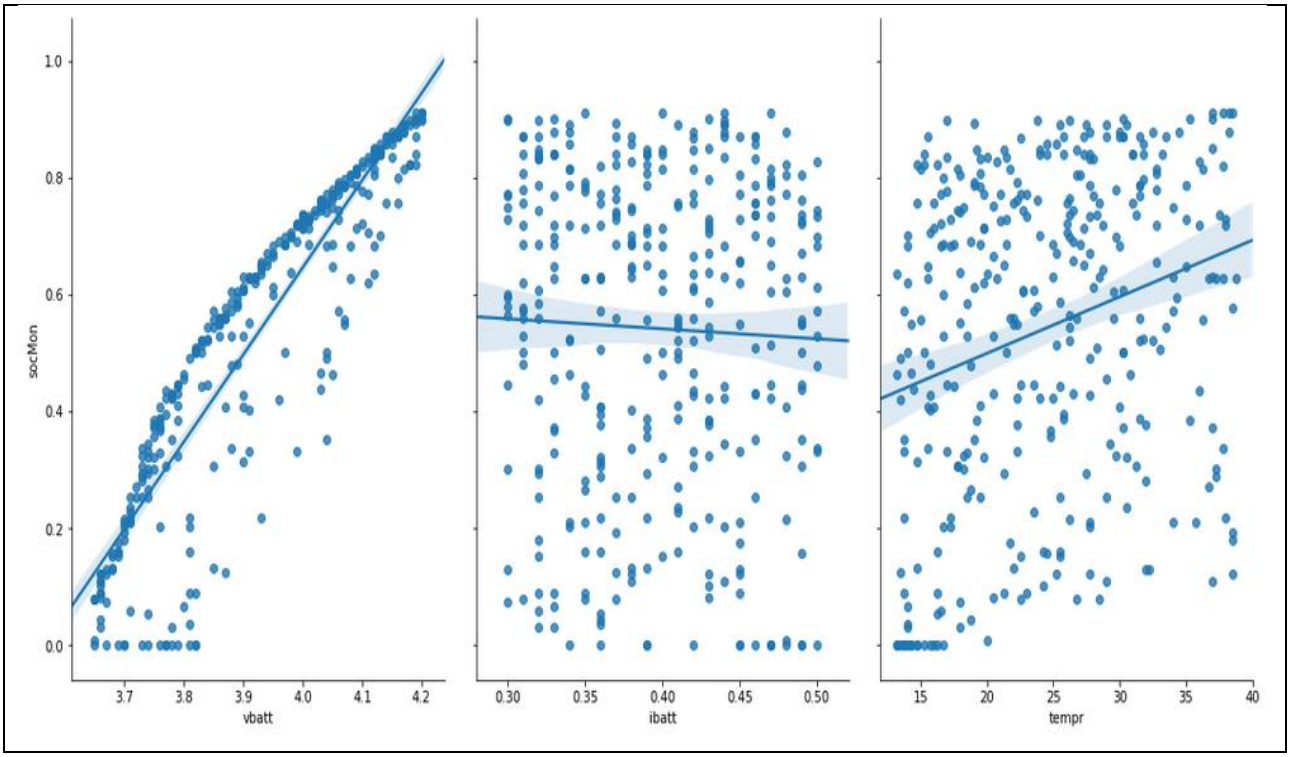


Fig. 4: In the plots below, a subset of the data is used. All data-points which satisfy the following condition are removed: (`ibatt < 0.3 & ibatt > 0.5`). It is seen that while the relation with vbatt and tempr are similar to the full-data plots, the relation with ibatt has become even more subtle. This gives an indication of the nature of non-linearity inherent to the SoC estimation problem.



Note: the subset used for this plot is only for initial exploratory purposes and not for model development.

Algorithms and Techniques

One of the industry standard methods to address regression modeling is by Perceptron based Neural Networks. The building blocks of such an NN are the perceptron, the learning rate combined with epochs (or iterations), an activation function, a pre-defined loss function, and an optimizer that minimizes the loss function.

A neural network is trained to arrive at an optimal set of weights and biases that minimize the loss. The activation function outputs a probability score – a floating point number between 0.0 and 1.0, as opposed to a discrete "yes/no" output. This fundamentally allows for neural networks to model non-linearities very well.

A Deep Learning model is proposed for this problem. It will be a Fully Connected MLP type. Given the size of the dataset and that MLPs are known to work well for regression problems, and the fact that the battery SoC estimation problem involves non-linearities, neural networks are believed to be the go-to model.

The following hyperparameters can be tuned during the search for the best model:

- Epochs
- Batch size – the number of data-points to use for one feedforward-backprop step in each epoch. Example: Say X_train=1000, and batch_size=10, there will be 1000/10=100 ff-bp iterations (of 10 samples) per epoch to arrive at the weights/bias set. For a given problem, it is an experimental (hyperparam search) process to arrive at an optimal batch_size.
- Optimizer – Stochastic Gradient Descent (sgd)
- Learning rate and Decay
- Momentum – to avoid local minima

Tuning of the Fully Connected network architecture:

- A wide vs. deep network
- Dropout – to avoid overfitting
- Weights initialization
- Activation: relu will be used because it is a continuous function; and given we are solving a regression problem, a continuous activation function is appropriate.

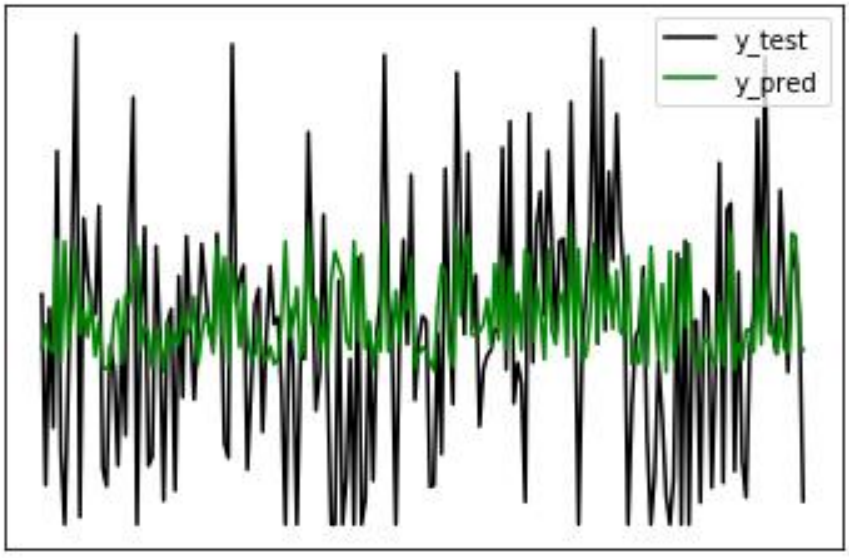
Benchmark

A linear model that uses the most correlated feature from Fig. 3 and Fig. 4 will be developed to model the target. The voltage at the terminals of a battery is the best indicator of the energy stored inside the battery. Scikit's linear_model.LinearRegression() was used for the benchmark.

This model resulted in the following metrics-

- MSE = 0.961 and
- R^2 = 0.066

Fig. 5 Benchmark model performance visualization



Note 1: Pre-processed data was chosen to develop the benchmark. Details about pre-processing are discussed next.

Note 2: Comparison of the benchmark and best model is discussed in the Results section.

III. Methodology

Data Preprocessing

The distribution characteristics of the data is discussed below-

Most of the data in our dataset is normally distributed – the median and mean are close to each other. This can typically be observed by plotting boxplots of the data.

Fig. 6: Data distribution

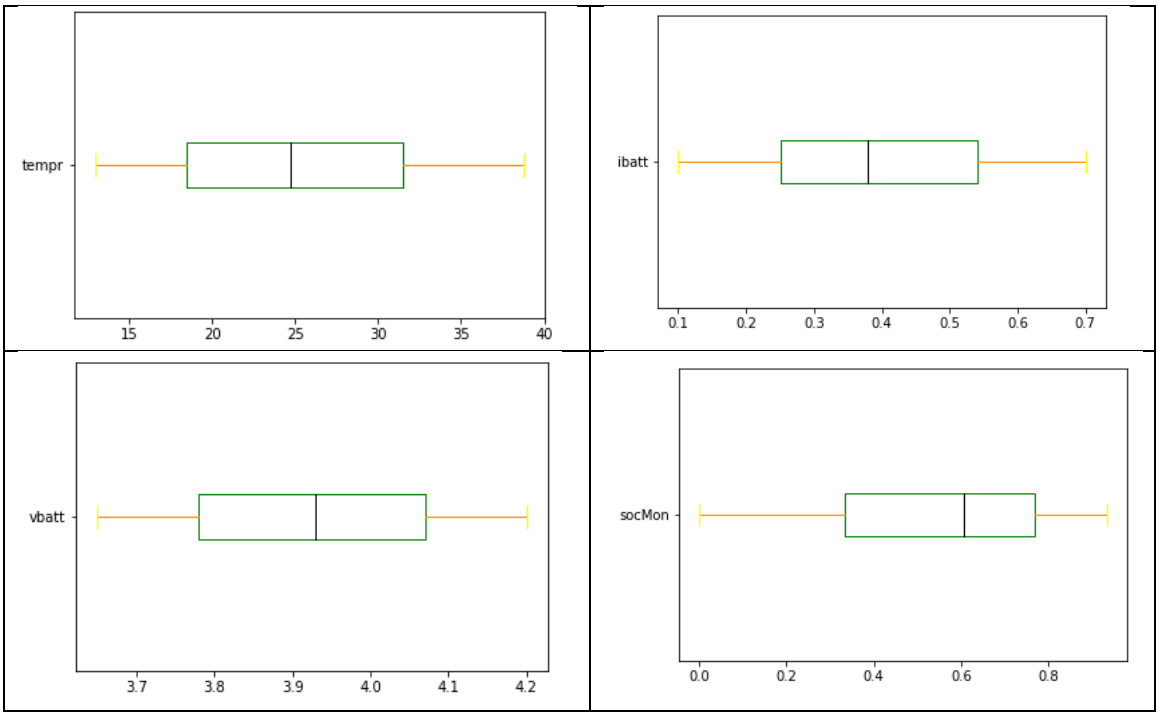
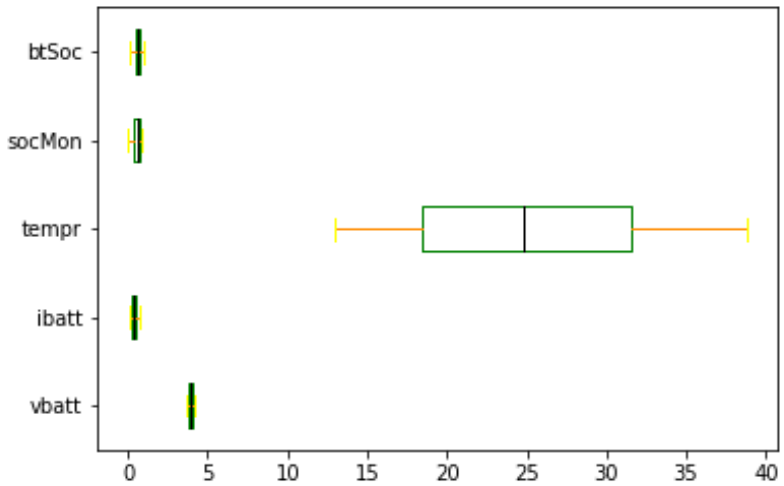
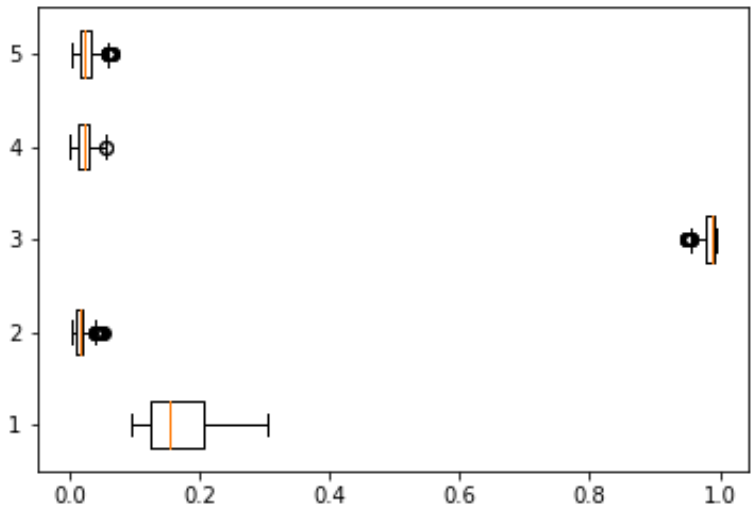


Fig. 7: Boxplots of all features and target (in a single view) – it shows varying orders of magnitude.



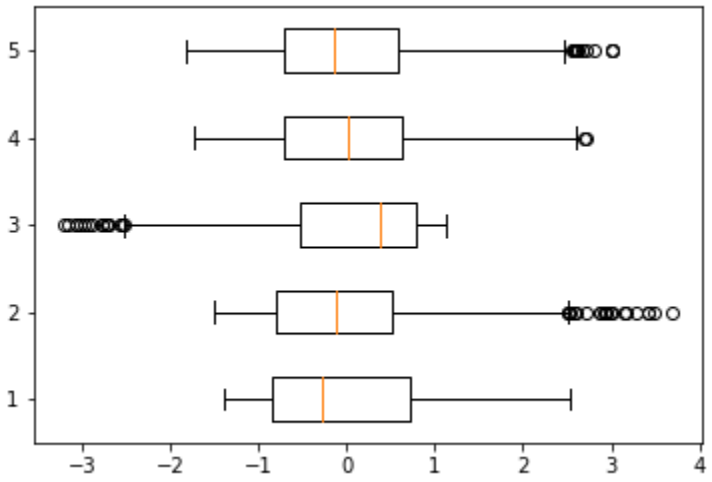
It is appropriate to normalize numerical data that lie in different magnitude ranges. ML algorithms perform better on data thus pre-processed.

Fig. 8: Boxplots of all features and target after normalization – all lie between 0.0 and 1.0



Scaling: this is also an important pre-processing step. This step will achieve the data to have a mean of approximately zero. An optimizer like SGD can converge to the minimum, more effectively on norm'd and scaled data.

Fig. 8: Boxplots of all features and target after normalization and scaling:



The normalized and scaled data is then split into training and testing set (20%). The resulting shapes as shown below:

```
X_train (804, 3)
y_train (804,)
X_test (201, 3)
y_test (201,)
```

Implementation

A Keras sequential() model with three hidden layers in addition to the input layer and output layer is built. The rectified linear unit “relu” activation is used for all layers, except the final layer. That is because, given the regression problem, the predicted output should be a numerical value without any transform. Dropout is added to every layer to prevent over-fitting. With dropout, a percentage of nodes will be inactive during feedforward-backprop steps of training – thus avoiding the possibility of a few nodes “controlling” the network.

It is valuable to initialize the weights and biases prior to training. The benefit can be understood by acknowledging the fact that if the weights are initialized to a very small value, then the signal gets far too attenuated as it passes through each layer, eventually becoming negligible. On the other hand, too large weights will cause the signal to blow up in magnitude. I have chosen weights initialization RandomNormal() with a mean=0.0.

The loss to be optimized is MSE and the optimizer is SGD.

Training is checkpointed – to save only the best model weights as training proceeds. The weights are stored in the `saved_models/` directory. These weights then are loaded after training is completed, to evaluate the model performance.

The training data is further split to result in validation data of size 10%. This validation data slice is unseen data during each training epoch. At the end of an epoch, performance (val_loss) of the model is evaluated on the validation data to conclude if an improved model has resulted in this epoch.

Fig. 9: Model summary

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_139 (Dense)            (None, 64)                256
_________________________________________________________________
dropout_105 (Dropout)        (None, 64)                0
_________________________________________________________________
dense_140 (Dense)            (None, 128)               8320
_________________________________________________________________
dropout_106 (Dropout)        (None, 128)               0
_________________________________________________________________
dense_141 (Dense)            (None, 64)                8256
_________________________________________________________________
dropout_107 (Dropout)        (None, 64)                0
_________________________________________________________________
dense_142 (Dense)            (None, 32)                2080
_________________________________________________________________
dropout_108 (Dropout)        (None, 32)                0
_________________________________________________________________
dense_143 (Dense)            (None, 1)                 33
=================================================================
Total params: 18,945
Trainable params: 18,945
Non-trainable params: 0
```

Note on the # parameters: as an example, the param# for the input layer is determined as follows:

- ([wt=64] x 3) + [bias=64] = 256. (64 is the number of nodes, 3 is the feature shape)

Coding gotchas

ML algorithms work with numpy arrays and not with pandas data-frames. As the data moves from the pandas world to numpy space, one has to watch out of subtle differences in doing the same things. Specifically, pandas data-frames have APIs to plot the boxplot; whereas with numpy arrays, I had to write my own python function (in `visuals.py`) with matplotlib to do the same.

I happened to be working with Jupyter (for the notebook) in firefox, as well as PyCharm CE 2017 2.3 (for pure python code such as `model_utils.py`) at the same time. When new functions were added to model_utils, they are not seen in the notebook even though all cells are re-run; they are visible to the notebook only after the kernel is restarted. In future, it may be a good idea to just use one IDE (jupyter or PyCharm) for development.

Coding support

I found the documentation on Keras.io/ very helpful to learn about implementation.

Refinement

Inferences of the model from the previous section resulted in a marked improvement in metrics from that achieved by the benchmark.

	Benchmark model	**Initial MLP**
MSE	0.961	0.340
R^2	0.066	0.669

For refining the model, the following best practices (industry standard) are applied:

Learning rate: "lr" is a small value, default = 0.01 for Keras's optimizers. This is used as a multiplier to figure out by how much to correct weights and biases during training epochs. The related decay parameter defaults to 0.0; which means that the learning rate is constant for all epochs. By allowing the learning rate to decay with time (epoch #), smaller changes to the parameters are made during mature training epochs, thus avoiding overcorrecting.

The following is used: `lr=0.01, decay=1e-6`

Batch_size: This is changed from the default of None to 8, which resulted in better performance than an increase to 64.

Num epochs: Increased to 500. Val_loss improvement was observed up until epoch #440

IV. Results

Model Evaluation and Validation

The refinement discussed above resulted in substantial performance improvement from that of the initial model:

	Benchmark model	Initial MLP	Refined model
MSE	0.961	0.340	0.054
R^2	0.066	0.669	0.948

A wide MLP network with just one 256-wide hidden layer was developed. The results of the wide vs. deep model were comparable to each other.

	Benchmark model	Initial MLP	Refined model	Wide model
MSE	0.961	0.139	0.054	0.061
R^2	0.066	0.865	0.948	0.941

A description of the final model is below:

- Input layer has, 64 nodes, three hidden layers 128, 64 and 32 nodes, single node output layer; Relu activation for all except last layer.
- MSE loss, SGD optimizer.
- Model trained for 500 epochs, (small) batch size of 8. A validation set of size 10%.

Regarding robustness of the model, there are two categories of unseen data – first the 10% validation split, and secondly the 20% X/y_test set. MSE loss on the validation data, `val_loss` is comparable to the training data `loss` for the final epochs as seen in the training logs. Further, the test data set metrics MSE and R^2 also indicate high quality predictions are made by the model.

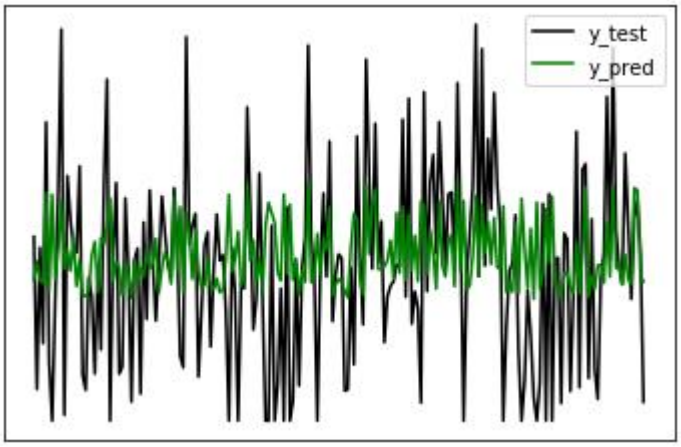
Justification

To have an appreciation for the robustness of the final solution, the following two points are to be noted:

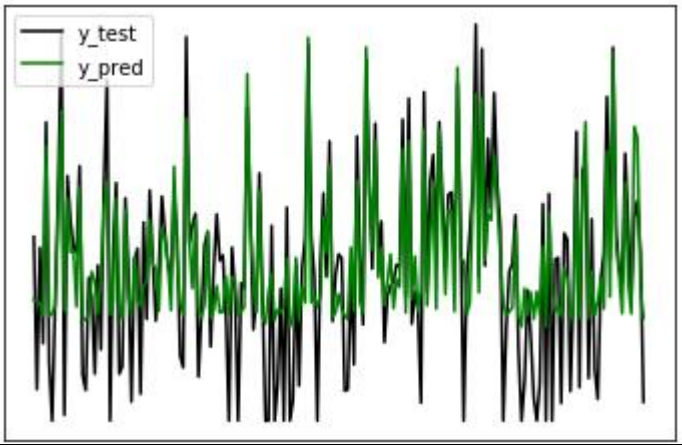
1. By going with two metrics for this project, namely, MSE and R^2, a high standard was set. MSE is a metric that is data dependent, and the R^2 score gives an objective score to the model. The final model high scores on both metrics.
2. The visualization below suggests that non-linearities between the 3 features have been modeled well.

Fig. 10: y_test vs. y_pred for the various models

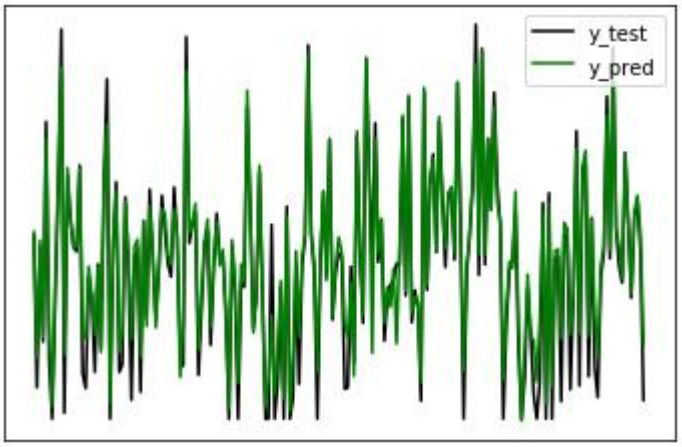
Benchmark model



Initial DL model



Refined final model



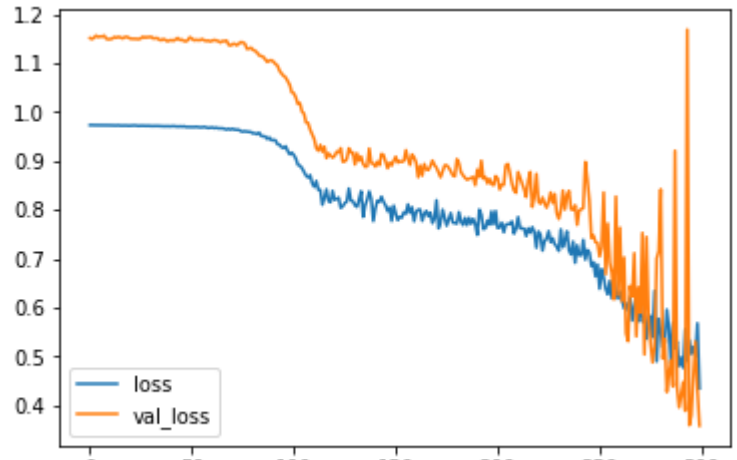
V. Conclusion

Free-Form Visualization

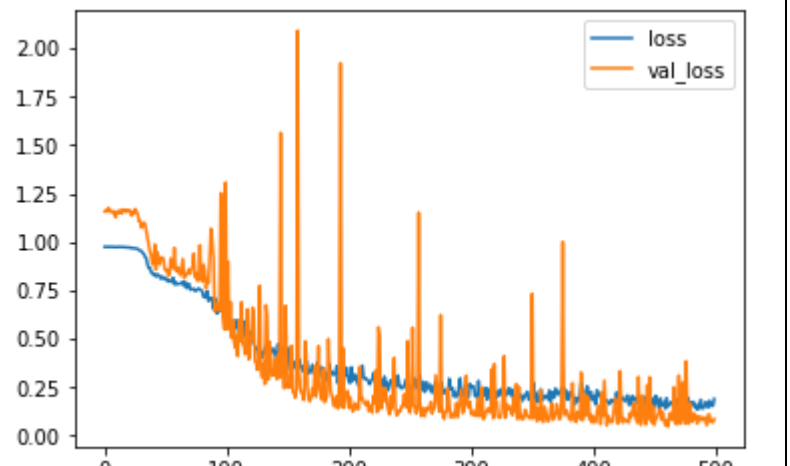
In the plots below, I compare the training history of the initial model with the refined model.

Fig. 11: Training history comparison

Initial model



Tuned model



The val_loss of the tuned model begins to converge towards training loss at an early stage. This indicates a better use of epochs. The reduced batch size for the refined model is the enabler for this improvement.

With the initial model, during later epochs, there seems to be overfitting happening frequently. This can be seen by the large val_losses around the 275+ epochs. On the other hand, in the tuned model logs,

this issue isn't so pronounced. It is likely that the small (dampened) learning rate applied during the late epochs is instrumental in controlling overfitting.

Reflection

Summary of process used for this project:

1. Stated an initial real-world problem, that is presently solved with dedicated hardware and not an ML/AI approach.
2. Generate the dataset by running hardware simulations.
3. Analyze the data. Apply pre-processing techniques to prepare the data for ML algorithms and frameworks like Keras.
4. Review available metrics and decide which is appropriate to use.
5. Develop a benchmark model.
6. Develop a DL based neural network.
7. Refine the DL model by tuning architecture and hyperparameters.
8. Review metrics for each model.

The area of battery SoC estimation is quite vast and I found it challenging to clearly state the problem in a manner that is bound and addressable for a capstone project.

As for the most interesting parts of the project, the model refinement steps that involved hyperparameter tuning was fun. I re-learnt quite a few things in the process of implementation. While writing this report, and during implementation of this project, I had to refer back to various projects I worked on as part of the MLND – it was nice to revisit them.

Improvement

There is the broader function of SoC estimation during normal charging or discharging of the battery. The work done in this project can be seen as the foundation being laid to achieve the broader scope.

The technique for hyperparameter search could be improved significantly (and in a more automated way) by using Scikit's GridSearch and Keras' wrappers for Scikit.

Since this DL model may find interest in mobile devices, MXnet which is a library that is better suited than TensorFlow for mobile platforms, may be a better option.