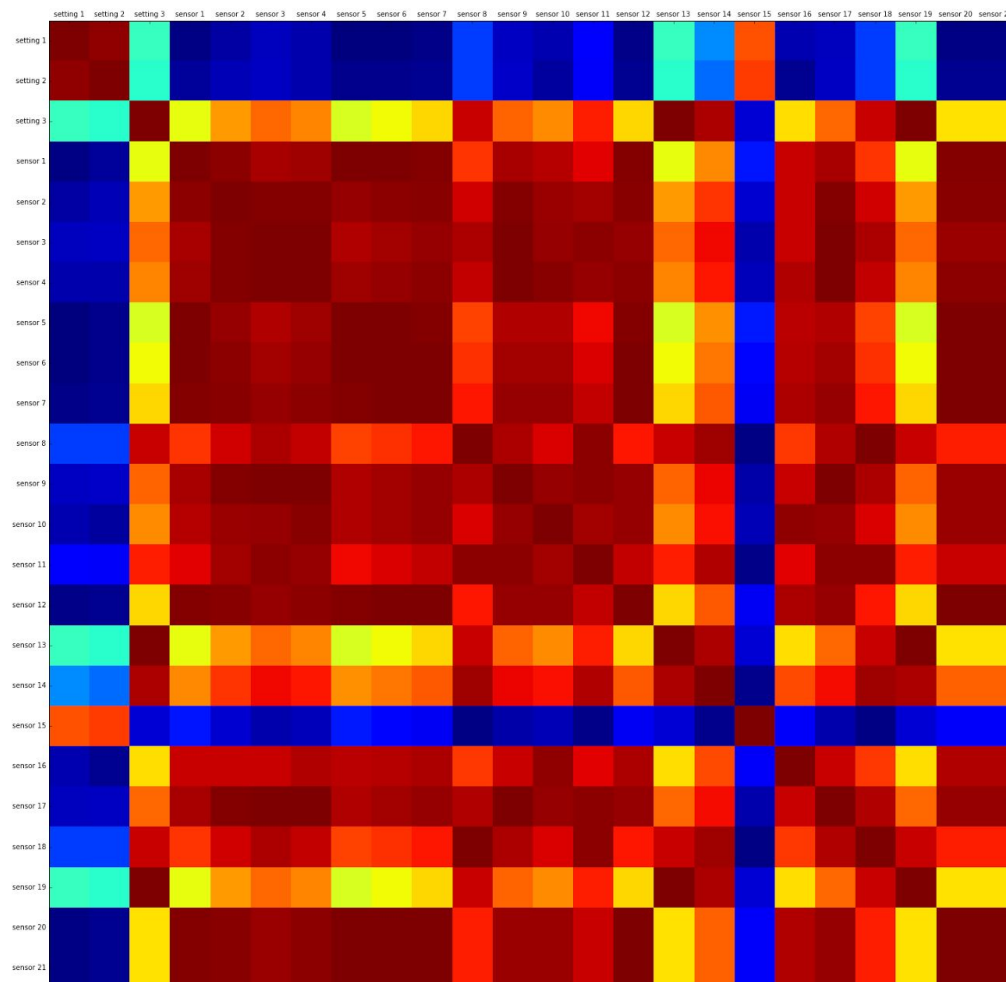


MODELING RESULTS

Before going into modelling, let's see how the time series of the raw test data are related to each other (code in the Assignment file).



Ponderings about Trivial Baseline Regressor

Why naive prediction wouldn't work?

Just knowing the engine ID to predict the remaining useful life of a test engine based on sensor features is not enough since what different datasets contain different engines that are independent of each other. If we had some similarity measure between the engines, then we could, but we don't. And if the client gets new engines, then they would also be independent of the previous engines at least by engine ID.

This means that the modeller needs features to make a better than a blind guess. Luckily, the client has provided the modeller with both historical stress testing results from sensors and settings in a form of multivariate time series together with test run sensor data.

Why RNN and LSTM -- Long-Short Term Memory network?

Let's start by exploring other possibilities:

Since we have a vector autoregression problem, it is natural to consider some stochastic processes based models, such as Gaussian Process Regression or Vector Autoregression developed in for example the STAN framework (RStan, PyStan).

However, since we live in the era of big data, it pays off to join the Bayesian statistics and Deep Learning efforts by utilizing a deep neural network, which in theory can model a process almost arbitrarily well, given enough data, by using the strengths of Bayesian statistics in the hyperparameter optimization subtask of getting a very good model (regression model in this case).

For this purpose, Gaussian processes can be used in the Bayesian optimization of the LSTM network parameters using the [GPyOpt framework](#).

As for why it has been have opted for to explore the RNN and LSTM direction, consider that

Recurrent neural networks (RNN) is a class of NN models that make use of the sequential nature of their input (in Keras, we start the model through `Sequential()`) such as speech or other type of time series data where the samples are (auto/cross)-correlated (dependent on the lag). A decent model for this type of data would be a Simple RNN cell. However, it turns out that One thing to realize is that a Simple RNN cell can be substituted by a LSTM model, where the only main difference (in addition to the amount of parameters!) is that robust to the vanishing gradient problem, which is important in deep learning. Robustness to vanishing gradient problem is very important for the gradient-based training process because that determines the learning speed and efficiency. Thus, one can opt to build a LSTM layers based model with good training options and robust and versatile outcome.

Training the LSTM Network

It is known that if U and W are the LSTM matrices for the gates, then the network has $4(nm+n^2)$ parameters. In any case, it seems a big number, as since so many hyperparameters have to be optimized, training a LSTM network may take a considerable amount of time.

If $U(n \times m)$ and $W(n \times n)$ are the matrices for the gates with dimensions, then the amount of total parameters is around $4(nm+n^2)$

Choosing the Parameters to Train

Due to limitations in computational performance, the following parameters were tested:

Time window length = {5, 18}, Dropout probability = {0.22, 0.5}, Units in LSTM layer1 = {32, 16}, units in LSTM layer2 = 20.

Why these parameters?

-- Batch Size: Usually larger batches are used for training than for predicting when 1-step predictions are needed. Decreasing batch size down to a low number makes the training more unstable and network weights change more often ($=1$ for online learning), yet makes possible 1-step predictions surely as well.

-- Time Window Size -- Classically, when having fixed data and samples, then in general, the smaller the window, the higher the variance and lower the bias of the estimator,

-- Training parameters:

Units -- There can be found some proof, a study by Baidu Research) recommending unit count of multiple of 32 when dealing with Float32. Other than that, higher unit count increases the training time substantially as well as the amount of parameters increases relatively rapidly.

Dropout -- the probability can be used to make the network be more robust to deletion of units, so that the learned representation from inputs to outputs is more versatile.

Optimizer type -- RMSprop, Adagrad (adaptive subgradient methods), SGD (stochastic gradient descent) or ADAM-family optimizers. RMSprop method was chosen, but there is more research done in the field, e.g. [this](#). Moreover, it is known that the configuration method of the optimizer could be more important than the selected routine.

Number of Hidden layers -- the more hidden layers, the more robust network can be learned. Mathematically they are all equivalent, but from topologic and practical considerations more deep networks are often preferred to learn more complex hierarchical representations. 2 LSTM layers were used in this case due to computational limitations.

Activation Function -- Tanh in LSTM layers, hard sigmoid in recurrent connections, Linear in output activation. There is not too much difference between tanh and sigmoid.

RELU vs Linear vs Sigmoid. It is clear that we cannot have softmax, since we have only 1 output cell. Sigmoid is not chosen because sigmoid is used for classification. RELU is not chosen because it gives only 0 output for negative feature values. Since we have performed minmax normalization and most features values are nonnegative (between 0 and 1), thus relu should perform similarly to linear activation function, but RELU would be a special case of the linear activation function: namely the one which passes the origin, thus we choose linear.

Results and Their Analysis

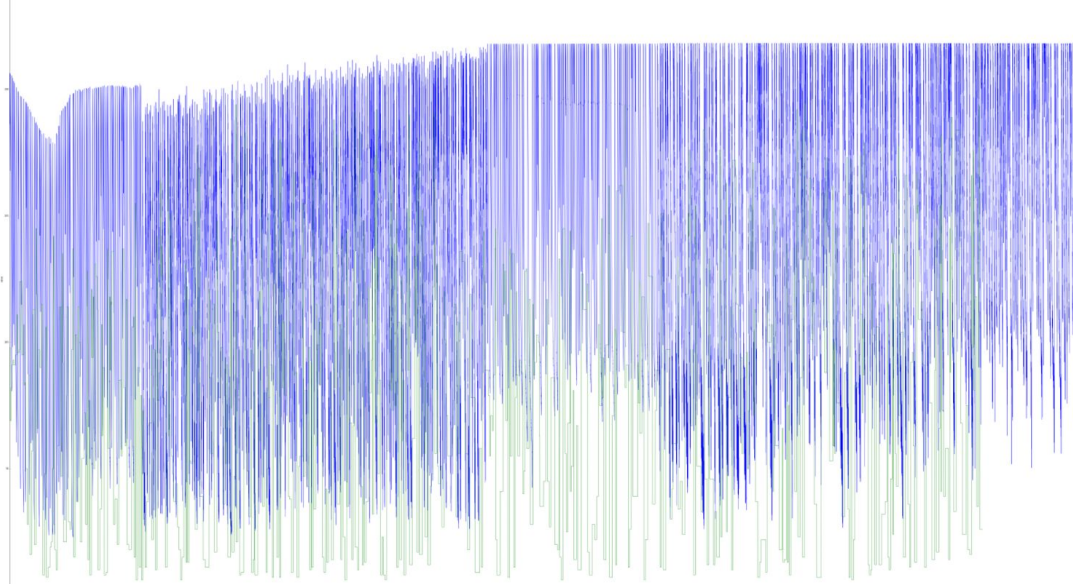
In the small simulation conducted on a home laptop, the winner was a RNN consisting of 2 LSTM layers trained on a time window data of size 18. No recurrent dropout as well as kernel regularizer used, the LSTM layers had dropout probability of 0.5. 20 units were used in the 2nd LSTM layer and 16 in the first one.

The used activation functions in the LSTM layers were tanh-functions and the used recurrent activations were hard-sigmoid functions.

The result is not surprising from the regards that given a relatively small LSTM network (16,18) can beat bigger networks when the training epoch count is relatively small (30). The strength of bigger networks comes up especially in case of huge datasets and long training time (high training epoch count). It also shows that indeed may pay off to generate longer time windows since in this way we generate more data for learning which can then be utilized.

To create a good visualization, the data should be subsampled to make it more sparse and then the predictions vs actual running useful lifetimes can be compared against each other visually.

Otherwise, plotting the results on the reduced test set results in a little bit too dense image:



Anyway, since the test error is practically the same as the (cross)-validation error, our LSTM model is definitely feasible and could be used to help our client to give feasible advice on the predicted remaining useful lifetime of any aircraft engine under consideration and when more computational resources can be utilized, the results can easily be improved a few margins at least.

By the end of the day, the result of the LSTM RNN deep neural network trained for 30 epochs is:














MSE: 4080.75 on the reduced test set

MAE: 54.68 on the reduced test set.

What else could have been changed?

Gradient clipping parameters, Cross-entropy cost function when using sigmoidal neurons, L1 or L2 regularization to the network.

Analysis of Running Times

 model_window18hidden16_20dropout0.5batch1.h5	15/01/2018 20:47	H5 File	70 KB
 model_window18hidden32_20dropout0.5batch1.h5	15/01/2018 20:35	H5 File	117 KB
 model_window18hidden16_20dropout0.22batch1.h5	15/01/2018 20:19	H5 File	70 KB
 model_window18hidden32_20dropout0.22batch1.h5	15/01/2018 20:06	H5 File	117 KB
 model_window5hidden16_20dropout0.5batch1.h5	15/01/2018 19:48	H5 File	70 KB
 model_window5hidden32_20dropout0.5batch1.h5	15/01/2018 19:42	H5 File	117 KB
 model_window5hidden16_20dropout0.22batch1.h5	15/01/2018 19:36	H5 File	70 KB
 model_window5hidden32_20dropout0.22batch1.h5	15/01/2018 19:30	H5 File	117 KB
 datasets_finalwindowlen8	15/01/2018 16:37	File	288,330 KB
 datasets_finalwindowlen5	15/01/2018 16:37	File	183,983 KB
 datasets_finalwindowlen15	15/01/2018 16:08	File	516,713 KB
 datasets_finalwindowlen10	15/01/2018 16:08	File	355,739 KB
 datasets_finalwindowlen18	15/01/2018 16:02	File	608,124 KB

We see that increasing the data size 239 % (corresponding to window size 5 to 18) the running time roughly increases by 100 % (from 6 to 12 min), so it seems clearly that the size of the hidden layer (and furthermore, the count of the hidden layers) increases the running time relatively more than adding the data (the rate of increase in running time is more strongly dependent on the network architecture than the input dataset), the dependence is sub proportional. This means that one should not be discouraged to generate a lot of features for the deep neural network and that is why the differentiated series could also be added.

Overview of Bayesian Optimization Experiments -- Lesson Learned

Bayesian optimization of the LSTM neural network was experimented using hyperopt's [fmin](#), but this function execution turned out to be too slow for practical applications. As an example, just to test one random probability value of dropout:

```
Params testing: {'dropout1': 0.2939556619766569}
Epoch 1/5
- 281s - loss: 12709.6188
Epoch 2/5
- 205s - loss: 7008.2583
```

In general it is noted that if a Mongo database can be used to execute the computation in parallel, a search space can easily be defined in a fashion as

```
space = { 'units1': hp.choice('units1', [32,64]),
```

```

        'units2': hp.choice('units2', [13,26]),

        'dropout1': hp.uniform('dropout1', .25,.75),

        'dropout2': hp.uniform('dropout2', .25,.75),

        'batch_size' : hp.choice('batch_size', [32,64,128]),

        'epochs' : 5,

        'optimizer':
hp.choice('optimizer',[ 'adadelata','adam','rmsprop']),

        'activation': hp.choice('activation',['linear','relu'])

    }

```

And by calling the fmin function on this search space should iterate through that space to find the best combination of hyperparameters

2), HYPERAS OPTIM was utilized. However, since hyperas optim uses fmin and fmin turned out to be extremely slow, this approach was neglected.

Since the first two obvious attempts at Bayesian optimization were too slow, manual optimization was opted for.

```

def base_minimizer(model, data, functions, algo, max_evals, trials,
                   rseed=1337, full_model_string=None, notebook_name=None,
                   verbose=True, stack=3):
    if full_model_string is not None:
        model_str = full_model_string
    else:
        model_str = get_hyperopt_model_string(model, data, functions, notebook_name, verbose, stack)
    temp_file = './temp_model.py'
    write_temp_files(model_str, temp_file)

    if 'temp_model' in sys.modules:
        del sys.modules["temp_model"]

    try:
        from temp_model import keras_fmin_fnct, get_space
    except:
        print("Unexpected error: {}".format(sys.exc_info()[0]))
        raise

    try:
        os.remove(temp_file)
        os.remove(temp_file + '.c')
    except OSError:
        pass

    try:
        # for backward compatibility.
        return (
            fmin(keras_fmin_fnct,
                 space=get_space(),
                 algo=algo,

```