For this challenge we had a lot of ideas, explored many of them and left out others, due to time constraints. We have mainly focused on trying to understand how to create a good model which could give good predictions, how to set the parameters for the model and how to handle the dataset in a way to maximize the learning curve of the model. In particular, for this last point, we have tried many different options (we will present only a few of them here).

We have not tried other loss functions other then mse and we have not tried to use other optimizers (we used only Adam which, on the paper, seemed to be the best). We tried a couple of times to use different activation functions and different learning rates (also using customized scheduling), but we did not go in depth with these ideas. Also, we have almost always used minmax normalization, even though sometimes we have also used standard normalization (or no normalization at all, just to try).

We have always tried to use a compact notebook (similar to the one seen in class), to easily run and train different models using just one click. In our `final` notebook we have placed useful functions to easily switch normalization method (standard or minmax). At last, we have decided to remove completely the test set because we have thought that using validation set would be good enough to our evaluation in the local environment (indeed, we wanted to have only a really rough estimate on how the model was performing, only to roughly understand if it was worth to submit or not).

Baseline models

As baseline model we have tried to implement a simple FFNN. However, for larger telescope size, as expected, the performances were rather poor (very noisy output) and we decided not to submit the model.

So we have decided to use the model seen in class as baseline. (`baseline_3.ipynb`) Even though, to be able to actually predict something and not just a mean we had to try few different parameters, after handling correctly the normalization and denormalization in the model file, we were able to obtain straight away the quite good result of `4.19` on the hidden set.

We have then tried to tweak this model using only LSTM rather then Bidirectional LSTM (just to make some trials and understand a bit better how the whole thing worked). We have also tried to stack multiple LSTM layers and to use a small dense fully connected output to the network. These trials have not led to better results in the local environment and we have only published on codalab few of them. However, this whole process was useful, both to do some trial and error, and to understand a bit better the 'right' dimensions of the `window`, `telescope` and `stride` params. Indeed, for our next models and next hypothesis we have kept them almost always (sometimes we have tried different params, such as `telescope=288` or different initialization) equal to `window=600`, `telescope=108` and `stride=10`.

After having a look at the data (we have computed the correlation between the various signals and actually look at the data) we have seen that there were some hint on what could be done. First of all, some sequences of data were rather strange: there were long sequences of ones which, to us, seemed like the data was corrupted; moreover, only a couple of sequences had high correlation (`Crunchiness` and `Hype root`) whereas, the other had particulalrly low correlation.

Multimodel approach

To exploit the correlation factor (`analysis_of_correlation.ipynb`) and because many of the models gave as output the mean of the input signal, we thought best to simplify the input to the model and to divide the

problem into more separate models. One model should compute: `Crunchiness` and `Hyperoot` given the two sequences, another one `Wonder level` and `Loudness on impact` and so on.

This, obviously, increased the training time by a lot (we wanted to train 5 models instead of 1) but we thought it could give nice results. However, we were proved wrong: the results were quite disappointing.

Handling dataset 'noise' and 'corruption'

As stated above, the long series of ones in the signals, which were present here and there in the dataset, make us suspicious of dataset corruption. Moreover, we really thought that it would have been better to smooth the data altogether; we thought: can high frequency features be really learned?

To handle the corrupted data and/or smooth the data we have tried really a lot of different approaches.

- remove the ones series by averaging the 'closer' points not set to one (`data_smoothing/smooth_with_window.ipynb`)
- using regression on each signal to smooth it out through low polynomials (we had to split each sequence in many sub-sequences and then to reconstruct the sequence: `data_smoothing/smooth_with_regression_4.ipynb`)
- using fft and ifft to remove only high frequencies
- using butter lowpass filter to filter out high-freq (`data_smoothing/smooth_with_lowpass.ipynb`)
- using autoencoder (for this approach we used a dense network: it was not the best idea, since the output was noisier then the input. We should have used an autoencoder with lstm. However, we did not try because we were already quite satisfied with the results of the regression approach) (`data_smoothing/smooth_with_autoencoder.ipynb`)

At last we have decided that the regression dataset was the best (we tried different values for stride and size)

We have also tried to handle the repeated samples and the corrupted zones of the dataset we handpicked the good parts, dividing the dataset in 7 sets, and smoothing them with some of the techniques seen above (`e1d1_conv_divided_set.ipynb`). The results were disappointing. We thought that the padding was the cause of the poor results, so we modified the `build_sequences` function to avoid the padding, but still the results were poor, so we abandoned this route.

Model

Following the ideas from the last challange, our first thought was to look for models to do transfer learning with: however not much about this can be found online, so we went on to looking for non-trained models. By trying out many different options, the models that we have found to work best both for training time and for the accuracy of the predictions have been `e1d1` and `e2d2`.

These models works with an encoder-decoder architecture, the second of the two including two of each. The `e1d1` model seems really effective to solve our problems. With a dataset smoothed with regression and by adding a convolution layer at the beginning we have been able to score the very good result of `3.74`. (`e1d1_conv.ipynb`)

We have tried then many variation on the theme: we have tried to use bidirectional lstm, we have tried to use skip connections and stacking different lstm blocks (modeled as encoder-decoder) over that architecture, switching to GRU cells (with and without conv layer) (`gru.ipynb`, `gru_conv.ipynb`), adding dropout (`e1d1_conv_drop.ipynb`), adding batch normalization, but without obtaining better resuts.

We also retried the aproach with multiple models using `e1d1` and regression smoothing and we got to a score of `4.05`. (`multi_model_2.ipynb`)

We tried to change the implementation of the `build_sequences` not to add padding at the beginning but to discard some points instead. Moreover, we have used a function to shuffle the sequences before training so that the validation set was randomly selcted. We knew that it could bring to overfitting, but we thought that for a prediction problem it could be an interesting approach, but instead it led to worse reults. (`e1d1_conv_alternative_val.ipynb`)

We have also noticed that on codalab the most important prediction to make to get a better overall rmse were on `Crunchiness` and `Hype root`. We have worked hard to improve these two sequences (again retrying the multimodel approach) by using `weights` for the input sequences (we have tried to multiply the normalized input sequences by the rmse output in codalab. We thought that with this we could have achieved different weighting for each sequences, thus we would have predicted better the highest weighted sequences, but actually we had no improvement...) and other techniques.
(`e1d1conv_tl_with_weigths_smooth_training.ipynb`)

We implemented, by looking again for models on internet, an encoder/decoder model a Luong attention layer, to help the prediction for the long input that we were using (`attention.ipynb`) and a version with a convolutional layer. Unfortunately, the results were disappointing.

At the end of the challange we decided to make an ensamble model of the two most promising models that we had, which were `e1d1_conv.ipynb` and `e2d2_conv.ipynb`, and make them predict only the feature that they were best with. From this attempt we got a result of `3.71` on the hidden set.

Final considerations

During the last days we have a little bit thought back at our assumptions to see where we might have been wrong: we were a bit disappointed to our positioning in the general ranking of the competitions and we wanted to improve. What we have come up with is that proably smoothing the dataset a priori was not the best idea. We have also tried to predict all the 864 samples in one pass.

To handle these problems we have come up with `no_smooth_final_lstm_skyfall_3`. Instead of smoothing the dataset we have stacked convolutions and maxPool layers to extract the high-level features from the data and then we have used a stacked approach of encoder-decoder lstms using one last dense layer (time-distributed) to get the final outputs. As usual in local we had quite good results but in codalab quite disappointing. we have also tried some variations on the sequences training to try to handle something like 'class imbalance' -> too many sequences are too similar so there is an overfit on them.

As last thoughts we think that our models were built by following good reasoning: we think that the problem was in how we handled the data and in how we handled overfitting. Indeed, in the last days we have also though that maybe dropout techniques and earlystopping in this problem might not be enough: it would have been better to use also weight regularization and batchnormalization. Indeed, we think that earlystopping was not the bets choice for this task, because of the difficult definition for a good validation set.

This challenge was particularly intriguing from our part because we have had much trouble to improve the predictions: we have tried many things but nothing seemed to really affect the prediction power of the network. We had to rethink all of our steps and to go back to revisit our assumptions and change them.