Artificial Neural Networks and Deep Learning - Challenge 2

Authors: Alessandrini Luca, Ronzulli Ruggiero, Venturini Luca

We faced this challenge by inspecting at first the different signals of the dataset given to us. We splitted the dataset with a separate train (training plus validation sets) and test set, in order to be able to test our results also in local. Before discussing the procedure we followed, it is good to explain how we divided the dataset into sequences for our trainings, and the metrics we used in evaluation. At the very beginning, we normalized each time series in order to have a range between 0 and 1 for every signal. Obviously, after the prediction, the results need to be de-normalized. We built sequences of variable length, called "*window*", starting from the training set. Starting from the first value of each time series, by sliding our window of a value, called *stride*, we generated our sequences with which we trained our models. To evaluate in an appropriate way our performance, we decided to use the *root mean squared error*, *rmse*, the one which is also used on CodaLab to return the scores.

After having noticed a periodicity of the signals, we started applying a very simple approach, similar to the one seen in class with two bidirectional LSTMs and their relative pooling. Initially, we tried to do both the direct prediction of all 864 values, and a prediction of the next 864 values, using the *auto-regressive* technique. It was immediately visible, that in every case the one with the auto-regressive technique was better: the shapes of the predicted portions of the signals, were directly comparable with the true ones, while the direct predictions were almost flat in many cases. Once determined this, we decide to focus on auto-regressive models, and we tuned the window and the stride used to divide the dataset in sequences. We did many attempts, and the one that gave better results, was a window of 90, and a stride of 5 (also 10 gave good results). After having determined the window and the stride, we tried to apply a gaussian noise layer in the model: we thought that this would have made the model more robust. Our hope was to create a better generalizing model, more independent from the noise of the signal. At first, we tried to set two gaussian noise layers, one between each couple of bidirectional LSTM layer, and pooling layer. This worsened the result. The results improved only when the standard deviation of the gaussian noise was close to zero, but still remaining a little bit worse than the model without the gaussian noise layer. This suggested to avoid putting further noise, because the signal was already quite noisy, and adding more, would have result in worse scores. Once determined this, we tried to implement different models, like with only "one-directional" LSTMs, or LSTMs with a different number of units. In the meanwhile, we thought that using models that mix CNN and LSTM (in cascade) could have helped in extracting some kind of "feature" of the signal and memorizing some kind of "trend". Unfortunately, even with parameters tuning and many attempts, these returned worse results.

Then, having seen the good results that the auto-regressive model gave to us, we tried to apply to this a "custom training", in order to try to perform a different evaluation of the loss function: instead of evaluating the loss function only on the next predicted point, we wanted to evaluate it on a little bit more points, like 5 in an auto-regressive way. Basically we train the network to predict a single following point but, during the evaluation of the loss function, we did some auto-regressive steps in order to predict some (e.g. five) following points. Then we compare these five predicted points with the real following five. We thought that this would have led to a more precise evaluation of the loss function. Even if in one of the seven signals this model was really good, in the others (and hence in general) returned a worse result with respect to our best model up to now. Since we were reasoning on the autoregressive technique from some time, we decided that it was time try to improve it a little bit: we focused on *Sequence to Sequence*. We knew that Sequence to Sequence was mainly used for translations, and is more suitable for text, but doing some researches on the internet, we found out that it was possible to adapt it also to times series prediction. We did few attempts, but already watching the local qualitative trend, the predictions were not satisfying.

After the lessons on the *Attention mechanism*, we immediately saw the big potential of this technique, and we decided to implement it. We read some articles on the Internet, and we found that there was a library called “`keras_self_attention`” ready to be used. We exploited its existence and we added the Attention to our previous models, and we tuned it: the better results we obtained were with only one bidirectional LSTM layer (128 units), one layer of Attention with sigmoidal activation. We tried also to invert the position of the bidirectional LSTMs and the layer of attention, varying both the number of attention layers and of bidirectional LSTMs layers, but this ended in worse results.

We also tried a more complex model (figure 1), composed of some parallel parts of the network in order to give to the convolution another try. It was composed of one input layer, which was the input of three separate parallel branches: two of them were exploiting convolutional, and implementing self attention; the other was only composed of a bidirectional LSTM and a self attention layer.

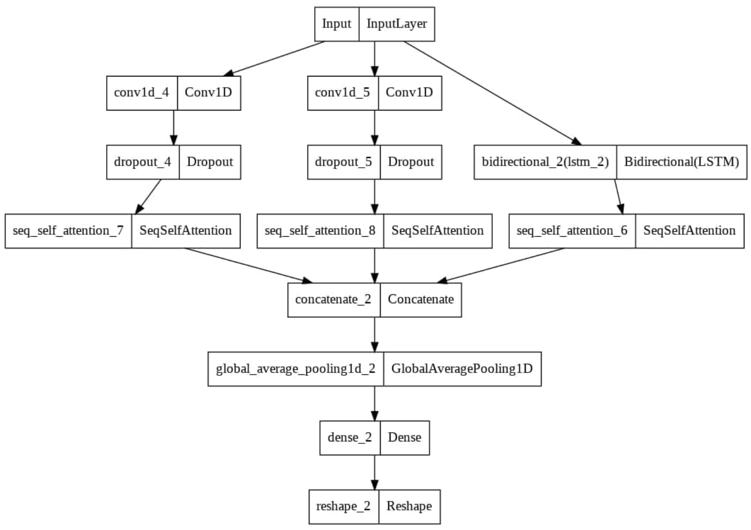
Input | InputLayer
conv1d_4 | Conv1D
conv1d_5 | Conv1D
dropout_4 | Dropout
dropout_5 | Dropout
bidirectional_2(lstm_2) | Bidirectional(LSTM)
seq_self_attention_7 | SeqSelfAttention
seq_self_attention_8 | SeqSelfAttention
seq_self_attention_6 | SeqSelfAttention
concatenate_2 | Concatenate
global_average_pooling1d_2 | GlobalAveragePooling1D
dense_2 | Dense
reshape_2 | Reshape

Figure 1: Model with different parallel branches.

Then, these tree branches were concatenated, and pooled. The idea of this was to merge the output of a CNN network and a bidirectional LSTM network: the difference with respect to one of our first models, with a CNN layer followed by a bidirectional LSTM one, is that here the two were in parallel: this in our mind could have solved the problems that we had before in stacking them. In this period unfortunately CodaLab servers were unreachable, but anyway, comparing the local results, this model was not noteworthy, hence we did not submit it. We focused then on the padding. Since we performed a padding with all zeroes, we thought that it could have been a good idea to try to ignore them when training our network: we implemented masks, that should be able to avoid feeding the network with the padding values in training. Also here we had to base ourself only on local results given by our local test set, due to CodaLab problems, but also this was not noteworthy. By looking at maximum and minimum of the seven signals, we noticed that there was a big discrepancy among them. Some series had a bigger range with respect to others: Since we were normalizing values before the training, the difference between them wouldn’t have been considered during the train phase, every signal could potentially have had the same range, and we thought that this could affect the result of training. Hence, we tried to train three different networks, each one with just a subset of all the seven signals as target. We divided the signals according to their range, and grouped the ones with similar ranges. After having written the three notebooks, we wrote the script that we used also in the `model.py` to merge the predictions. We did also another attempt, dividing all seven signals in seven “groups” of one signal each, but in none of these two cases we had an improvement, and the results were close to the ones we obtained training all of them together: probably, this happened because the training phase already coped this difference by tuning the weights.

We also found that a stateful LSTM could have improved our results: we tried to implement that. We faced some problems in setting up the notebook, because the LSTM stateful requires a specific size for the batches, and the number of input values must be divisible by the batch size, otherwise an error was thrown after the first epoch. In order to cope with this constraint, we discarded some sequences from our dataset. The results were good, comparable with our previous better results. We tried also to change the window size, but this produced again worse results.

We realized that maybe our manual tuning could have some limits, and we tried to exploit "*Keras Tuner*" on our models. This module trains the same model setting automatically different hyperparameters values, on the base of the results after some epochs the better values are displayed. We realized that we have not considered the option of the hyperbolic tangent, as suggested by Keras Tuner: hence we tried all the changes proposed by the automatic tuner, but the results didn't show any substantial difference.

We tried then to change completely approach, by implementing a *Moving Average* model. This model, studied in other courses, could help in predicting time series: for example this model can be really helpful in trading, when trying to predict stock markets. The concept seemed similar to our problem: we wanted to predict time series. After having implemented a Moving Average of order one, we realized that the prediction shapes were promising: unfortunately, this required a library not present in the docker of the server. The library was too complex and had too many dependencies to be replicated manually, hence we decided to explore other approaches. We tried to preprocess the data in order to create smoother time series. To do this we applied a filter that weighted the values of a point with the values of the other nearest two points, but we didn't find any improvement with this change.

At this point we decided to exploit other knowledge acquired from other courses, and we focused more on data. After having observed again the time series shape, we thought that a good way to proceed could have been the to decompose the signals into three parts: **trend**: the increasing or decreasing aspect in the series; **seasonality**: the periodicity component extremely visible in our series; **noise** (the split is visible in figure 2). The first two are the two fundamental ones, the ones that we wanted to train the network (since noise, in theory, should be not predictable). The better thing to do at this point, was to train our model only on trend and seasonality: hence we took the seasonality component, and we added it to the trend component. We then trained our best model up to now (the one with attention mechanism, window of 90, stride of 5, and a 128 units bi-directional LSTM, with a result of $4,12$) on "seasonality + trend". This model increased a little bit our performances: we obtained $4,05$. We tried to change again some hyperparameters, like batch size, window and stride, but were not able to beat our first score: even with a small change, the model could result in worse scores. Analysing again the distributions of the data, we realized that the first 20000 samples, often had a different shape with respect to the rest of the sequence: we decided to try removing this part (even if it is a considerable part of the entire dataset). In any case, this worsened a little bit the result, probably due to the smaller training set. Since then we have noticed that the dataset was not as heavy as the one of the first challenge, and because of that the training was quicker, we decided to implement K-Fold cross-validation: in this way, we exploit the knowledge of all the dataset, leaving one fold out of training at each iteration. We tried different values for K, on our best three models: the best model up to now, the one trained with trend and seasonality; our second best one, the one with attention mechanism, window of 90, stride of 5, and a 128 units bi-directional LSTM; our third one, similar to the second one, but with 256 units in the bi-directional LSTM layer. Cross-validation helped in reducing all the scores, bringing them below 4: our best score after this idea, was near $3,45$.

At this point, we noticed that different models, were good in predicting different series: we tried to mix our models, by exploiting the strengths of each one for some specific time series. For each series we took the model which returned the best result, and we combined all these models in order to increase our score. By doing this merging we obtained our best result, which was near $3,35$.

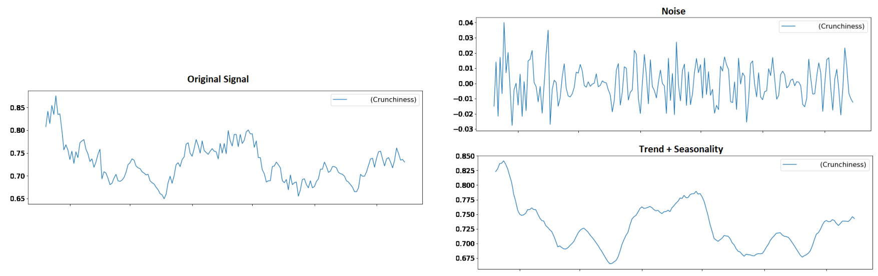

Figure 2: On the left the original signal, on the right the decomposition of it. The noise is not a perfect white noise, but close to it.