The problem

In this challenge, we were asked to predict future samples of uncorrelated time series given in input with a fixed length of 200. The goal was to implement a forecasting model that leverages past observations in the input series to predict future values. The task requires the model to exhibit generalization capabilities in the forecasting domain, i.e. it is required a model that is not limited in predicting in a single or predefined time domain.

During the development phase, we mainly focused on trying different **approaches** regarding what the model learns from the time series, maintaining the model architecture fixed. Below there is the description of this architecture which, as we observed, worked fairly well.

Data Preprocessing *[1]*

Our first step has been making data pre-processing. Firstly, removed the padding from each time series using the two indexes that we could find in the valid_periods file and grouped them in the six different categories.

Then, we proceeded to analyze the time series. We collected some general info about our dataset:

- The length of the time series ranges widely between 24 and 2776, with the average length being around 200. This info slightly changes within each category;
- The range of the values within each time series has been already normalized between [0, 1];
- Each category contains a variable number of series. The last one (F) contains a very small amount of time series with respect to the others.

By plotting random time series for each category, we observed that they don't show any particular shared behavior (This may be due to the fact that they are unrelated between each other).

However, a lot of them seem to have a periodic and trend behavior which means we were not dealing with stationary time series. As explained later, one of the used approaches considers this peculiarity to model the future of the time series.

The next step has been to think about how to make the split in training, validation and test set.

Firstly, we considered dividing every time series in three slices, one for each set, but then, due to the poor results we were getting, we decided to insert each time series in one and only one set. For each category we considered 90% of the time series for the training set and 10% for the validation. Consequently, we also decided to remove the test set.

Lastly, with respect to the categories, we followed two different approaches. At first we thought about building one model for each category and appropriately switching the used model inside the predict function of the model.py, depending on the categories of the input time series. In a second moment, we considered all time series together without distinguishing among categories. In the end, the latter performed better, probably because the amount of the training data in the last category was too small to generalize well and this negatively affected the overall MSE.

The model *[2]*

In all the different approaches, that are listed below, we used the same network and hyperparameters.

We set the batch size at 64 and the epochs at 200. The choices for the layers of the network have been the following:

- A masking layer, which is a way to tell sequence-processing layers that certain timestamps in an input should be skipped when processing the data;
- Three Bidirectional Long Short-Term Memory (LSTM) layers. The first one has 256 units in each direction and by setting return_sequences = True, it returns the full sequence to the next layer. The second one is similar but with 128 units in each direction. The third layer has 64 units and return_sequences = False, this because the resulting model is a sequence-to-vector in which you use only the output of the last time step.
- A fully connected layer with a number of output neurons equal to the telescope (9 during the Development phase and 18 in the Final phase);

The model is then compiled using MSE as the loss function and the Adam optimizer for adjusting weights.

Regarding the callbacks, we decided to use EarlyStopping with patience equal to 15 as a way to limit overfitting and ReduceLROnPlateau with factor 0.9 and patience 3 to speed up the training.

The build_sequence and compute_sequences_for_dataset functions

We decided to expand the number of time series, by implementing the functions *'build_sequences'* and *'compute_sequences_for _dataset*'. With the first one we work on a specific series and split it into multiple blocks of length 'window'. Each block is composed of a x part of length 'window' – 'telescope' and a y part of length 'telescope'. 'Window' is the length of the input of our network, the 'stride' is the number of samples to skip before starting the next window. The *'compute_sequences_for_dataset'* function gets in input the whole dataset and it calls the *'build_sequences'* function to increase the number of samples. 'Data' is a list containing the starting series and 'minimum_length' is the minimum length that a series must have to be considered. With this function we also handled the padding of each series which was

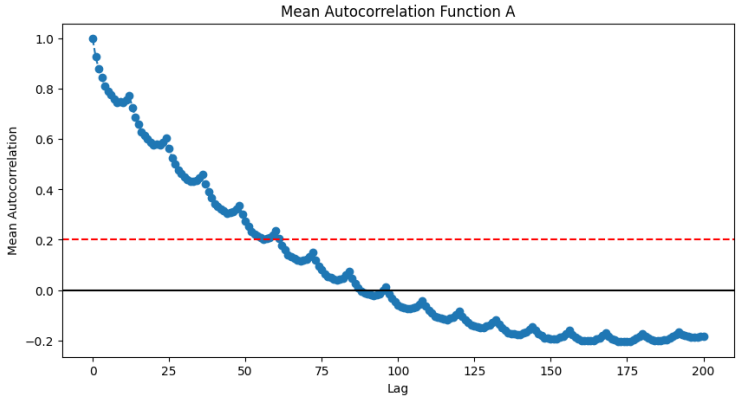
smaller than the window but longer than 'minimum_length'. More precisely, the last 'telescope' points of the series are positioned at the end. Right before, we inserted a padding with a value of 2 and the rest of the initial part is given by the rest of the series.
We decided to use a stride much smaller than the window in the training set to have more time series available for the fit of the model. Instead, we set the stride equal to the window in the validation set to avoid having an optimistic estimation of the validation MSE.

The following are the different approaches used to transform the time series used to build and train the models. Each approach uses the same model just described but different types of time series.

First approach of handling the time series (only using original series)
The first approach has been to consider the original time series without any modification of their values. The 'compute_sequences_for_dataset' function was used to expand the number of time series and the two approaches on the categories of the time series were implemented as mentioned in the last paragraph of the data preprocessing. Given these settings, we then proceeded in developing the network and training the model. Finally, we computed the MSE and MAE for the predictions.

Second approach of handling the time series (only using detrended series or only using stationary series) ***[3] [4]***
In the previous approach the model learn the ways to predict the future given a generic time series as input. These time series can be very different between each other, having completely different behaviors. Consequently, the network is asked to learn and model all these different behaviors so that it can successfully generalize and predict the future of an unknown time series. In the second approach we tried to simplify this problem by considering only the stationary part of the time series, so that the model didn't need to worry about the trend and the periodic behavior of the time series.

Firstly, we plotted the mean of all the autocorrelation functions, each associated to a time series. This mean is shown for the time series of the category A in the following figure (as a function of the lag, i.e. the number of time units of which we shift the time series to correlate it with itself).



By observing the plots for different categories and different lags, we noted a common (on average) descending trend of the autocorrelation functions, with occasional periodic spikes. This finding gave us the suggestion that on average each time series has a **trend**, which is the reason why the autocorrelation function descends towards 0, and a **seasonality**, which gives explanation to the periodic spikes.
For these reasons we thought about an algorithm which :

- First, remove the trend and the seasonality from each time series, getting as a result the corresponding stationary time series;
- Then use the function *'compute_sequences_for_dataset'* , defined in the previous paragraph (First approach), to divide all the result from the previous step in features (the X) and targets (the Y);
- Then train a model with this features(which are stationary) and targets;
- And finally evaluate the performance, in terms of MSE, by taking the predictions of the stationary equivalent of the time series in the validation set (the stationary features), adding back the seasonality and the trend, and confronting the result with the original targets (not the stationary ones).

When the model is in production, each time series given as input is first preprocessed by removing its trend and seasonality (exactly as it's done for the time series in the validation set), obtaining the stationary features. Then those features are given to the model, which outputs the stationary predictions of the future. Finally those predictions are brought back in the original domain by re-adding the seasonality and the trend.
With this approach the training is faster and the network is more precise, because it is required to learn only how to model the stationary component of a given time series (the residual part).
We follow this discussion by specifying the methods used to remove the trend and the seasonality from each time series, and the methods to add them back when making predictions.

The pipeline we used to obtain the stationary series corresponding to a time series is logically described by the following formulas :

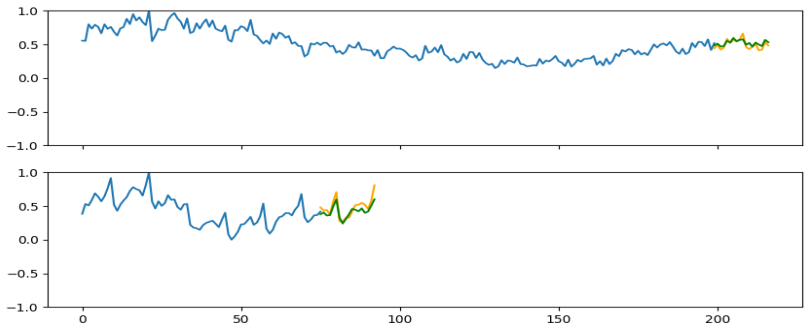
ORIGINAL_TIME_SERIES - TREND = DETREND_TIME_SERIES
DETREND_TIME_SERIES - SEASONALITY = STATIONARY_TIME_SERIES

Among the different methods that can be used to remove the trend from a series, we chose to remove it by removing from *ORIGINAL_TIME_SERIES* another series which is obtained considering a **rolling mean**, with rolling window = 5, of the elements of the *ORIGINAL_TIME_SERIES* (with padding on the left).

To remove the seasonality from the *DETREND_TIME_SERIES* we first identified what is the period of the time series by using the autocorrelation function of the *DETREND_TIME_SERIES*. If there is a seasonality, this autocorrelation function will be similar to a periodic function and so identifying the first peak (excluding the first few lags, lag = 0,1,2,..) would be a simple way to identify the period. We then use the technique of **differencing** to remove the seasonal component from *DETREND_TIME_SERIES*, by subtracting from it a lagged version of itself, with lag equal to the period we just identified. For the reverse transformation (going from stationary to original), which needs to be done during predictions, we added back these components with a similar reasoning. The technical details are present in the notebook provided together with this report.

Interestingly, the performances obtained (validation MSE) with this approach are worse than the ones obtained with the first approach. One of the reason could be that not every time series has a seasonality component (thing we proved by observing also the plot of the mean autocorrelation function of the other categories and by plotting some time series) and so both the method to identify the period and the differencing technique discussed earlier could have provided us with non-stationary time series in the end. This, of course, would have negatively affected the model performance.

Indeed, by redoing all this reasoning, but without considering the seasonal component (so only working and learning with *DETREND_TIME_SERIES*) we obtained a model with slightly better performance than the one obtained in the first approach. The model in this case is trained with time series features which can have or not a periodic behavior (seasonality), but that don't have a trend. This puts the model in the middle in terms of complexity, if we consider the model of the first approach (most complex, because it has to learn also the seasonal and trend behavior of the time series) and the model who learns on stationary series features (less complex).
The following is a plot showing the comparison between predicted and true values of the future values of some time series for our best model, the one trained on detrended series.



In the end we chose to submit an ensemble of two models, both trained on detrend time series but with different dataset splits in training and validation, so that the most of the dataset would have been explored by all the models. This approach could have been repeated in order to obtain better performance by generating more than two models, each trained on a different subset of the dataset, following a similar strategy to k-fold cross validation.

Other experiments: *[5]*
We also focused on models utilizing one-dimensional convolution layers. The first encountered issue was related to padding. After conducting some research, we discovered that convolutional networks don't perform well on padded datasets. Consequently, we proceeded to overhaul the entire data pre-processing, ensuring that all time series had a length of 209 for the first dataset and 218 for the second. This adjustment allowed us to avoid padding and optimize performance. While testing the model, we also observed a significant deterioration in network performance when using a Cropping1D layer with a range from 200 to 9/18. To address this, we decided to incorporate a MaxPooling1D layer after every convolutional layer, excluding the output layer. This modification enabled us to crop only from a length of 25 to either 9 or 18. As a final improvement, we introduced a bilateral layer before every convolution to extract information from the past prior to each convolutional operation. To summarize, the structure of the network was as follows: input -> bidirectional_1 -> Conv1D_1 -> MaxPooling1D -> bidirectional_2 -> Conv1D_2 -> MaxPooling1D -> bidirectional_3 -> Conv1D_3 -> MaxPooling1D -> Conv1D_output -> Crop1D.