Hands-On Sequence Prediction for Time Series

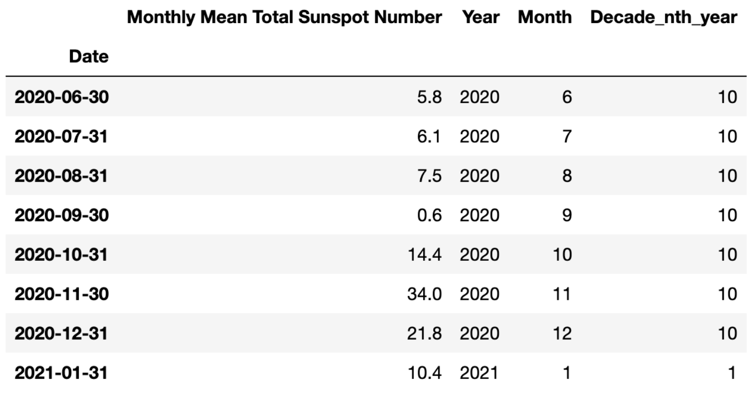
Marcel Pons Cloquells

June 17, 2021

1 Introduction

In this section, different Deep Learning Sequence Prediction Models for time series will be performed upon the **Sunspots Dataset**, which contains temporal data consisting on 3265 entries going from years 1749 to 2021 about the monthly mean total sunspot activity, a temporary phenomena on the Sun's photosphere that varies across time and follows an approximately 11-year cycle.

The dataset provided in Kaggle does not need any preprocessing since there are no missing data and neither outliers. Since the dataset contains a *Date* column, it is considered as a `datetime64` datatype when importing, in this way any time series manipulation performed in the future will be easier.



	Monthly Mean Total Sunspot Number	Year	Month	Decade_nth_year
Date				
2020-06-30	5.8	2020	6	10
2020-07-31	6.1	2020	7	10
2020-08-31	7.5	2020	8	10
2020-09-30	0.6	2020	9	10
2020-10-31	14.4	2020	10	10
2020-11-30	34.0	2020	11	10
2020-12-31	21.8	2020	12	10
2021-01-31	10.4	2021	1	1

Figure 1: Structure of sunspots dataframe, where the *Date* column had been considered as the index.

Before diving into sequence modeling, an brief introduction of time series together with a exploratory data analysis will be necessary to understand the data.

Time series is typically defined as an ordered sequence of values that are usually equally spaced over time. Depending on the number of values that are in each time step, we can talk about univaritate or multivariate time series, being the later conformed by multiple values at each time step. The sunspot dataset consists on an univariate time series, where the single value that changes across time is the sunspot activity, as it can be seen in Figure 2 below.

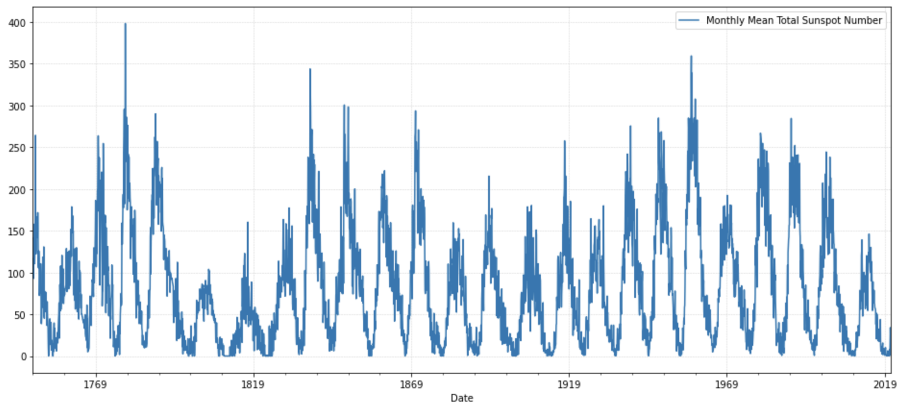


Figure 2: Plot of the mean monthly sunspot activity across time (1749-2021).

As it can be appreciated in the figure, the sunspot activity presents *seasonality*, which is a typical characteristic of many time series consisting on patterns that repeat over time. More specifically, this pattern coincides approximately with the 11-year solar cycle. However, the repeating pattern is not regular over the cycles, where some of them present much higher peaks than others. If two consecutive cycles are plotted jointly over the 11 years that they last, this seasonality can be further appreciated.

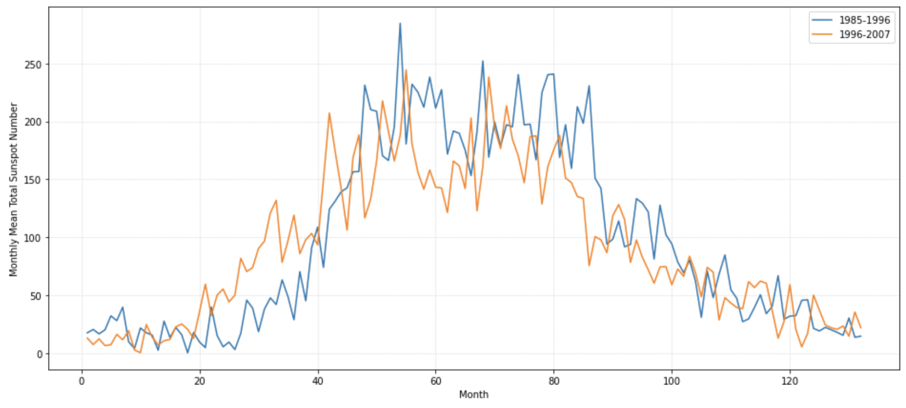
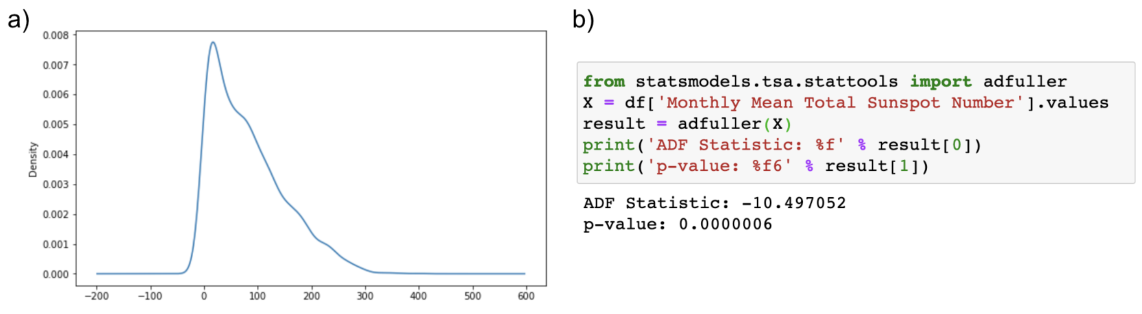


Figure 3: Sunspot activity of years 1985-1996 and 1996-2007 represented on the same axis of 132 months.

Another pattern typically found in time series data is the *trend*, which is observed when the values have a tendency to decrease or increase over time. The sunspot data seems not to follow a general slope conforming a trend, since the sun’s activity does not increase or decrease above all across the years (i.e. the new cycles do not increase or decrease but they are constant over a range of values).

Stationary time series are those that its statistical properties, mean and variance, do not change over time. The *stationarity* of a time series is an important aspect to take into account when analyzing them, specially when carrying out statistical analysis, the majority of them working on the assumption that the data follows this characteristic.

Stationarity can be checked by visualizing the density distribution of the data, which should conform a Gaussian shape. In Figure 4 we can see that sunspot data distribution presents more or less a bell shape, however the values are skewed to the left, thus it is difficult to state with high confidence whether the data is stationary. There are some statistical tests that are more robust and informative about whether data is stationary or not, such as the *Augmented Dickey-Fuller test*, which determines how strongly a time series is defined by a trend, process that varies the mean and variance over time, ultimately affecting stationarity[1]. The null hypothesis H_0 of the test denotes non-stationarity (a time-dependent sequence), whereas the alternative hypothesis H_1 denotes stationarity.



```python
from statsmodels.tsa.stattools import adfuller
X = df['Monthly Mean Total Sunspot Number'].values
result = adfuller(X)
print('ADF Statistic: %f' % result[0])
print('p-value: %f6' % result[1])
```

```
ADF Statistic: -10.497052
p-value: 0.0000006
```

Figure 4: Assessing stationarity. a) density distribution of the data b) Augmented Dickey-Fuller (ADF) test.

As it can be seen, the *p-value* of the test is far lower than 0.05, therefore rejecting the null hypothesis and concluding that the sunspot data is stationary. That is can be considered an important fact because stationary time series are easier to model and analyse statistically[2].

There are several statistical techniques to analyze and forecast time series, such as *Moving Averages*, *Exponential Smoothing* and *Autoregressive Models*. However, Machine Learning Models are frequently used in time series for modeling and predicting, achieving considerably good results in many fields (economics, weather forecasting, medical signals, etc.).

2 Modelling

Data Preparation

In order to apply Machine Learning algorithms to time series, the data has to be prepared, dividing it into features and labels. The features (input x) are a number of values in the series, often known as window, whose size is a potential parameter that has to be taken into account when applying the models. The label is the immediate entry/value that follows the window. A visual representation of a window and the following label is illustrated in the figure below.

[1]The seasonality of time series also affects the stationarity. Given the results of the test, we see that the seasonality of the data previously seen is not too strong to make the statistical properties non-constant over time.

[2]In the cases that time series are non-stationary, techniques for converting it to stationary are usually applied, such as differencing and log transform.

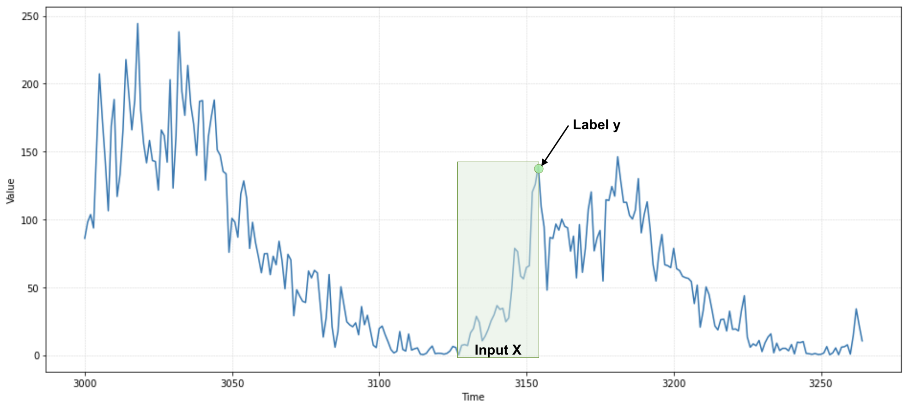


Figure 5: Representation of a window of size 30 months (square) and is label (circle). The data corresponds to the validation set of the sunspot dataset that will be used.

Three models are going to be trained on the sunspot data: a dense neural network (DNN), a Recurrent Neural Network (RNN) and a LSTM Neural Network. Tensorflow (`tf`) and Keras are going to be used to implement them. The architecture between models will be intended to be as similar as possible in such a way that that comparisons between models will be based as most as possible on the type of network used. To do so, the architectures will have a similar number of layers, the same metrics and optimization algorithm, as well as they will be trained with the same number of epochs.

The models will be fed with this prepared windowed data and will learn how to predict the next value given a sequence of n window size. Therefore, the sunspot dataset saw previously (Figure 1) has to be transformed into a windowed dataset compatible with the tensoflow models. The `windowed_dataset()` function is defined to make these transformations.

```
def windowed_dataset(series, window_size, batch_size, shuffle_buffer):
    dataset = tf.data.Dataset.from_tensor_slices(series)
    dataset = dataset.window(window_size + 1, shift=1, drop_remainder=True)
    dataset = dataset.flat_map(lambda window: window.batch(window_size + 1))
    dataset = dataset..map(lambda window: (window[:-1], window[-1]))
    dataset = dataset.shuffle(shuffle_buffer)
    dataset = dataset.batch(batch_size).prefetch(1)
    return dataset
```

The first line of the function creates a tensorflow dataset object, which is then passed to the `window()` function. This function is the responsible of creating the windows, whose parameters are: `size` which controls the size of the flat elements generated; `stride` and `shift` which determine the stride and the shift of the input elements, respectively; and `drop_reminder`, which truncates the data by dropping the possible remainders that generate after the last window is generated. For example, if the function is passed to a sequence of range from 0 to 7 represented in Figure 6, by specifying `size=3, stride=2, shift=1` and dropping the reminder, the result would be 3 sequences conformed by the 3 number that the arrows cover, not using the last reminding 7. If `shift` is increased to 2, the resulting first sequence would be like `[0,2,4]`.

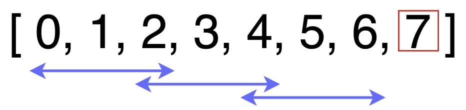


Figure 6: Schematic representation of sequence windowing.

The second line using `flat_map()` ensures that the order on the dataset stays the same. Then in the third line the split separating features from labels is generated with `map()`, where every sequence is split to all its values (`[:-1]`) but the last one (`[-1:]`), which will be the label.

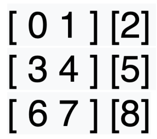


Figure 7: Schematic representation sequence split.

Up to this point the sequences are prepared to be inputed to the tensorflow models, however the last two lines are recommended for better performance. Since many times the order of things can impact the training of the models (sequence bias[3]), the order of the sequences is randomly shuffled with the `shuffle()` method[4]. Next, the sequences are wrapped into batches of `batch_size` in order to make the training of the models to take several sequences at a time instead of only one, thereby speeding up performance. Finally, the `prefetch()` method makes the batches of data ready to use while the current one is being processed.

Splitting the data and applying window function

Once the window function is implemented, it can be applied upon the sunspot dataset. Firstly we take X and Y from the dataset, corresponding time (month number starting from Jan 1749) and the sunspot activity. Then X and Y are split into training and validation set, being the former the first 3000 months and the later the 264 most recent reminding months (22 years) (the validation has been visualized previously on Figure 5). Next the `windowed_dataset()` function is called specifying the window size, `batch` and `buffer` size. The window size is a relevant parameter that affects the performance of the model, for this reason, five different sizes will be used and compared when analysing the results. The resulting dataset will be the input of our neural network models.

```
X=df['Index'].values
Y=df['Monthly Mean Total Sunspot Number'].values

split_time = 3000
time_train = X[:split_time]
x_train = Y[:split_time]
```

[3]The favouring of the items due to their position in a sequence, where the ones at the beginning and at the end have a tendency of being selected more than those in the middle.

[4]The buffer takes n elements from the dataset and it is used for large datasets that do not fit in memory, but also is used to speed up the random shuffle.

```
time_valid = X[split_time:]
x_valid = Y[split_time:]

window_size = [12, 30, 60, 120, 132]
batch_size = 32
shuffle_buffer_size = 1000

dataset = windowed_dataset(x_train, window_size, batch_size, shuffle_buffer_size)
```

Dense Neural Network

The first architecture implemented is a dense neural network conformed of two hidden layers with `relu` activation function ($f(x) = max(0, x)$), the first one with 20 neurons and the second one with 10. The output layer is a single neuron returning a final unique prediction for each windowed input. The metric used to assess how far the predictions are from the true train labels is the Mean Square Error, which penalizes more the errors (1). The loss will be used to calculate the gradient, which will be passed to the Stochastic Gradient Descend algorithm (`SGD`), to optimize iteratively the weights and biases of the network[5].

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{1}$$

Once the model is compiled, we train it calling `model_fit()` using 100 epochs, which denotes the number of complete passes the model takes through the training dataset.

```
# Dense Neural Network
model = tf.keras.models.Sequential([
        tf.keras.layers.Dense(20, input_shape=[window_size], activation="relu"),
        tf.keras.layers.Dense(10, activation="relu"),
        tf.keras.layers.Dense(1)
])

model.compile(loss="mse", optimizer=SGD(lr=1e-8, momentum=0.9))
model.fit(dataset,epochs=100,verbose=0)
```

Dense Neural Networks consider the inputs as fixed, therefore they do not handle ordered sequential data because they do not have a notion of time or sequence. Despite that, by taking a sufficient amount of training data, they can learn characteristic time series patterns that sometimes lead to good predictions.

[5]Instead of taking steps by computing the gradient of the loss function $J(x)$ generated by summing all the loss functions like Gradient Descent does, SGD takes steps by computing the gradient of the loss of only one randomly sampled example, thus avoiding getting stuck and converging faster.

Recurrent Neural Network

As it was seen in the last delivery, Recurrent Neural Networks maintain the inherent relationship in the sequence relating the computation that the neurons are doing in a particular time step from prior steps as well as the input at the current time step. RNN are implemented by using the Keras `SimpleRNN` layer, which takes as input a three dimensional tensor with shape `[batch, timesteps, feature]`. Therefore, it is necessary to expand the dimensions to 3 (at the moment batch and timesteps are only taken) by applying a lambda layer calling `expand_dims()` to every input `x`, and setting `input_shape=none` to take sequences of any length. In this way, having a window size of 30 timesteps batched up in sizes of 32, the shape will be `(32, 30, 1)`, being 1 the dimensionality of the input at each time step since we are working with univariate time series.

Once the dimensions are compatible with the RNN, the first layer will learn from the windowed sequences, passing a hidden state from one step to the other and adjusting its weights. On each step the cell will get the current input and the previous output. For this layer, `40` units are specified, being a tunable parameter that denotes the dimension of the inner cells of a recurrent neuron. Moreover, `return_sequences` is set to true in order to return a sequence of outputs instead of a single output, thereby this sequence can be the input of the following RNN layer (i.e. it serves to stack one RNN on top of another).

Since the default activation function is the hyperbolic tangent ($f(x) = \frac{e^{x}\breve{}e^{-x}}{e^{x}+e^{-x}}$), its output values range between -1 and 1. For this reason the output Danse layer will return values conditioned by the previous RNN layers. Due to the fact that the outputs of the sunspot dataset range from 0 to 400, it is recommended to help the process of learning by scaling up those final outputs of the Dense layer. To do so, another lambda layer is added scaling the final outputs by a factor of 100.

```python
# Recurrent Neural Network
model = tf.keras.models.Sequential([
  tf.keras.layers.Lambda(lambda x: tf.expand_dims(x, axis=-1), input_shape=[None]),
  tf.keras.layers.SimpleRNN(40, return_sequences=True),
  tf.keras.layers.SimpleRNN(38),
  tf.keras.layers.Dense(1),
  tf.keras.layers.Lambda(lambda x: x*100.0)
])

model.compile(loss="mse", optimizer=SGD(lr=1e-8, momentum=0.9))
model.fit(dataset,epochs=100,verbose=0)
```

LSTM Neural Network

Long-Short Term Memory Networks are similar to RNN, but they contain different interacting layers called *gates* that control which information passes through the network and which not. Moreover, a cell state goes through the sequence as a flow of information, being effectively and selectively modified in each time step by the gates, consequently having longer memory. That is, the data from earlier timesteps in the window can have a greater impact on the overall prediction than in the case of RNN. Due to this characteristics LSTMs are considered better than RNNs when predicting time series.

A LSTM neural network using two Keras `LSTM` layers is implemented in the same way than the previous RNN, since they need the same input dimensions and use the same activation functions.

```
# LSTM Network
model = tf.keras.models.Sequential([
  tf.keras.layers.Lambda(lambda x: tf.expand_dims(x, axis=-1), input_shape=[None]),
  tf.keras.layers.LSTM(38, return_sequences=True),
  tf.keras.layers.LSTM(38),
  tf.keras.layers.Dense(1),
  tf.keras.layers.Lambda(lambda x: x*400.0)
])

model.compile(loss="mse", optimizer=SGD(lr=1e-8, momentum=0.9))
model.fit(dataset,epochs=100,verbose=0)
```

Comparisons

In order to obtain the results of the predictions, we iterate over all possible windows, starting from time `0` to the time of the last possible window (`len(Y)-window_size`) and for each window `model_predict()` is called to make the prediction of the next value (`np.newaxis` is called to expand the dimensionality to 2 for future compatibilities). All the predicted values are stored into a list, which is sliced to obtain the values of the validation set. Finally all array predictions are stored together in a 1 dimensional array.

```
forecast = []
for time in range(len(Y) - window_size):
    forecast.append(model.predict(Y[time:time + window_size][np.newaxis]))

forecast = forecast[split_time - window_size:]
results = np.array(forecast)[:, 0, 0]
```

With the results of each model (and trying different window sizes on each one), the final performance of the model for predicting the validation set is calculated by using the Mean Absolute Error between real and predicted values ($\frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$).

```
tf.keras.metrics.mean_absolute_error(x_valid, results).numpy()
```

Window Size	DNN	RNN	LSTM
12	13.9007	15.2418	14.0384
30	14.5918	15.3188	15.8590
60	15.6462	15.1561	**13.5877**
120	17.0593	15.0249	14.3933
132	17.3368	17.6033	17.6034

Table 1: MAE of the models using different window size.

Table 1 contains the results for the different models with different window sizes. It can be noticed that Dense Neural Networks, despite not recognising the notion of sequence and considering the data as fixed over time, perform surprisingly well even with a simple model like the one implemented. The underlying idea of these results is that this type of networks discover characteristic patterns that help them make good predictions. Besides, it is also noticed that the performance results are better with small window sizes, and it decays as they get bigger. This may stem from the fact that DNN do not learn relations and dependencies over time periods, thus increasing the size only increases the fixed input data without considering the possible relations that can be detected over longer time periods.

On the other hand, Recurrent Neural Networks, expected to give better results than DNN because they maintain the relationships of values in the sequence, give the worse predictions among the other models. What it can be noticed is that this time the results only decline with the biggest window size. In fact, using a window size of 120 months (10 years) returns the best results, denoting that the network really captures new relations with more time steps.

Finally and meeting the expectations, LSTM networks bestow the best results of the three models, being 60 the optimal window size and declining when going towards the extremes sizes 12 and 132. This demonstrates that by using gates that dynamically select which information is forgotten, updated and kept, but also passing a cell state that stores longer term information, the models learn considerably well the inherent structure of the sequence and perform good predictions.

If the best predictions obtained by the LSTM model are plotted together with the real values of the validation set, it can be appreciated that the forecast approaches significantly well the sunspot activity, capturing the increases and decreases. Some peaks and noise are not captured by the model, but this is expected because we want models that generalize well on unseen data, not overffiting and learning the noise of the training data.

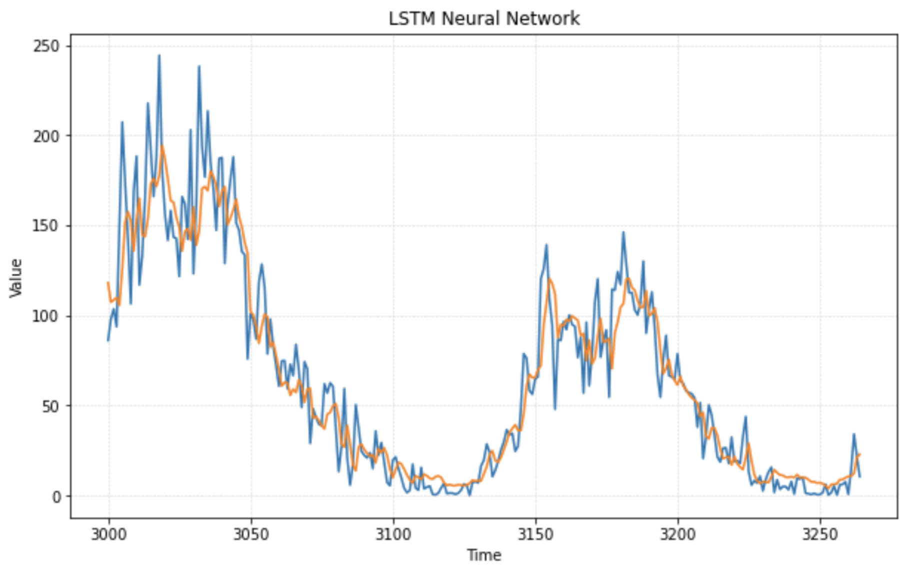


Figure 8: LSTM predictions (orange) vs actual values (blue) in validation set.

3 Conclusions

Different neural network architectures were tried for predicting time series data about sunspot activity. Initially, some exploration of the characteristics of the series have been undertaken. Then, we went through the process of preparing sequence data in order to make it compatible for deep learning models, and realized how the window size is a key factor on the final performance of those models[6].

The model that gave the best predictions was a Long-Short Term Memory neural network conformed of three layers, however a simple Dense Neural Network surprisingly generated good predictions, approaching those of the LSTM and overcoming Recurrent Neural Networks.

Further architectures could have been tried, such as increasing the number of layers and units in each one, trying different optimization algorithms (like RMSProp and Adam), adjusting the learning rate, making models combining LSTM layers with DNN layers, or even extracting characteristic features using Convolutional Neural Networks. Nonetheless, simple models oftentimes get good enough results, even surpassing more complex ones like we saw in DNN against RNN.

[6]The complete codes used for this report are available on the attached HTML.