DETECTING EVENTS IN TEMPORAL LOG DATA

National Research University Higher School of Economics
Moscow, Russia

DIMAS MUNOZ MONTESINOS
SUPERVISOR: ILYA MAKAROV

May 11th, 2020

ABSTRACT

Time Series Classification (TSC) has been considered as one of the hardest problems in data mining. With the exponential increase of data availability (in particular, time series data) a wide variety of TSC algorithms have been proposed, although only some Deep Neural Network can solve this task. This growth also means that we depends more on automatic classification of time series data, and ideally, we should be able to apply algorithms with the ability to work in large datasets.

Although in the past, common approaches using RNNs to solve TSC did not succeed, nowdays this complex task is feasible. We show an overview of some Neural Networks which are successful in this problem.

CONTENTS

1 INTRODUCTION

The amount of data generated rises exponentially everyday. This fact has opened the door for more research areas in the field of data science, and time series data analysis is certainly a niche which could still be explored so much.

Time Series Classification (TSC) is not a new phenomenon: it is one of the major research areas over the past few years mainly due to its large number of practical applications in various domains. For example, in the context of quantitative finance the primary goal of time series analysis is forecasting, in the context of signal processing is used for signal detection and estimation, in the context of pattern recognition can be used for clustering or anomaly detection.

In the context of this project, we use TSC to split our data into two categories, that is, a binary classification. The time series come from different sources: the main dataset is collected by some sensors during the excavation of boreholes, and the second dataset consists in engine noise meauserements.

The first dataset (which we are going to call *Boreholes Dataset* or *BD*), as we see in the further sections, is not "ideally" collected, thus there are many issues that we must fix before we can use the data. It is very likely that we cannot obtain good results from such dataset, hence we use a second dataset[1] (which we will call *Ford Dataset* or *FD*) that has been collected in optimal conditions.

In the next sections, we use some terms that are defined here:

Definition 1. A univariate time series $X = (x_1, x_2, \ldots, x_T)$ is an ordered set of length T such that $X \in \mathbb{R}^T$.

Definition 2. A M-dimensional time series (MTS) $X = (X^1, X^2, \ldots, X^M)$ consists in M univariate time series such that $X^i \in \mathbb{R}^T$.

Definition 3. A dataset $D = \{(X_1, Y_1), \ldots, (X_N, Y_N)\}$ is a collection of pairs (X_i, Y_i) where X_i could either be a univariate or multivariate time series with Y_i as its corresponding one-hot label vector.

Given the previous definitions, we can write our problem more formally: *the task of TSC consists of training a classifier on a dataset* D *in order to map from the space of possible inputs to a probability distribution over the class variable values (labels)* [7].

The TSC tasks were commonly solved by using RNNs (Recurrent Neural Networks). However, they use mainly sequential processing over time, which means that long-term information is processed sequentially through all cells. The information can be easily corrupted because of it is multiplied many time by small numbers close to 0 (this problem is known as *vanishing gradients*). Hopefully, now we choose from a wide variety of Neural Networks which are successful in this problem, such as LSTM[1], GRU or SVM to mention a few ones.

Note that it is easy to find similarities from this scenario to computer vision problems, where algorithms learn from the spatial information contained in an image. Essentially, here we have the same class of problem, but in one only dimension (univariate time series).

To deal with TSC, we use different technologies:

- Python 3.7 to preprocess the data and train the model(s).
- PHP 7.4 (which runs behind a nginx server) to manage the content of a web-based application for data view.
- PostgreSQL 12.1 to store the data.

1 The Ford Dataset is available at http://www.timeseriesclassification.com/description.php?Dataset=FordA

Additionally, we use many third-party libraries in our scripts, such as Tensorflow[2], Scikit-Learn[3], Numpy[4] and Pandas[5] among others.

In an effort to simplify the installation/configuration of the mentioned technologies, we use Docker[6] containers during all the development.

2 DATASETS

2.1 Description

On the one hand, the BD consists in 11 GB of data of exported SQL data from Postgresql[7] and 2.4 MB of data divided in several CSV files. On the one hand, the SQL files contain the data collected by each sensor in the time: position of the driller, pressure, temperature, spindle speed... On the other hand, the CSV files contain information about:

- The properties of the sensors: sensor name and the units of time series.
- The experiments with the starting and finish times. Each experiment consists in a series of boreholes.
- The boreholes with the starting and finish times.
- The layers and phases of each borehole, which include diverse measures, such as *material*, *burr height in* and *burr height out* among others.

On the other hand, the FD consists of 500 measurements of engine noise. The data set were collected in typical operating conditions, with minimal noise contamination.

Despite the FD is already cleaned and ready to work with, the BD is not yet usable. Before we start working with it, we need to clean and transform BD into simpler items.

2.2 Preprocessing

2.2.1 *Boreholes*

There are some boreholes in the dataset without start and stop time. We could not locate these boreholes in the time, thus we have removed them since we cannot find their associated sensor data.

2.2.2 *Phases of boreholes*

Each borehole is composed of some phases: a borehole can contain zero, one or more phases. Each phase has a date range and many attributes which are obfuscated. Furthermore, some of these attributes might be empty, which make even more complicated to interpret the data.

If the params of a phase are empty, then we assume that the value has not changed since the previous phase. Hence we sort the phases by time and then we inherit the last known value(s).

During this process, we face another problem: some phases do not contain information about the time. We cannot sort them in the time, thus it is not possible to inherit the value of the previous phase. In such case, we just ignore them.

2 https://www.tensorflow.org/api_docs/python/tf
3 https://scikit-learn.org/
4 https://numpy.org/
5 https://pandas.pydata.org/
6 https://www.docker.com/get-started
7 https://www.postgresql.org/

2.2.3 Sensor data

We clean the data of sensors as follows:

- We remove all the zero values.
- To clean non-representative values from time series, we define that each value in the time series must be "close" to at least other 4 values in a time window of 30 seconds (otherwise, we remove it). For example, if there are 4 values and, one minute later, there are 10 values, then we remove the first 4 values. Doing this, we remove single values in the sequences.
- We take the time windows of all boreholes and remove the sensor data which is not in any of those windows.
- The sensors are collecting many values per second. In order to simplify the large time series, we compute the average of all values per second and store it.

In other words, the dataset D_s of a sensor s is defined as a set of pairs (X_i, Y_i) where every X_i is a univariate time series such that $x \in \mathbb{R}_+$. Also, informally, it must satisfy that every $x_k \in X_i$ is surrounded by another 4 values from X_i.

Recall that each time series in the same dataset D_s may have a different length.

2.2.4 Sensor events

To process the data more efficiently, we put an effort to locate all time windows where there is data. Here we introduce the concept of *sensor event*:

Definition 4. A sensor event E_s (in a sensor s) is described as the maximal time windows in a time series $X \in D_s$ such that the separation between two consecutive values $x_k, x_{k+1} \in X$ is less than a certain value α:

$$x_k, x_{k+1} \in E_s \Leftrightarrow |x_k - x_{k+1}| \leqslant \alpha$$

In our dataset D_s, we fix α to 30, which can be interpreted as all sequences where the separation between two consecutive values is less than 30 seconds. For example, if we find a group of 10 values and, one minute later, we find another group of 15 values, then we could say that there are two events in the borehole since they are separated by more than 30 seconds.

Despite some boreholes have a generous amount of sensor events, there are some boreholes which does not contain any sensor data (thus they do not have any sensor event).

2.2.5 Classes of boreholes

Based on some criteria, we assign a label/class to each borehole: *good* or *bad*. This step is a must before we can apply any supervised machine learning algorithm.

The *good* boreholes are seen as follows:

- It must happen that *roundness mean* is less than 0.06 and *roughness ra mean* is less than 1.6.
- If *material* is *Aluminium*, then it must happen that *burr height in*is less than 1.127 and *burr height out* is less than 1.127.
- If *material* is *Titanium*, then it must happen that *burr height in* is less than 0.2032 and *burr height out* is less than 0.2032.
- If *material* is *CFPR*, then all these values must be empty (i.e. *null* or *false*): *delamination in*, *delamination out*, *breakout in*, *breakout out*, *chip residues*.

Similarly, we label boreholes as *bad* as follows:

- It does not have a *quality card*.
- It has a *quality card* without measures. For example, if we know the material but the *burr height* values are empty, we labeled it as a *bad borehole*.

Some boreholes contain parameters not measured (i.e. *null* values). In such cases, we just use the criteria that can be applied (in the *non-null* values) and assume that the rest (the *null* values) is correct.

2.3 Analysis of datasets

Firstly, we analyze the Boreholes Dataset as it is the "main dataset" in this project.

After we do all the preprocess steps, we find that the total number of good (931) and bad (905) boreholes is almost balanced (the difference is insignificant: $\sim 2.8\%$). However, if we filter these numbers to boreholes with sensor data, the result is excessively asymmetric: 177 good boreholes and 98 bad boreholes (see Figure 1). This implies that there is a difference of 44.63% between good and bad boreholes with sensor data.

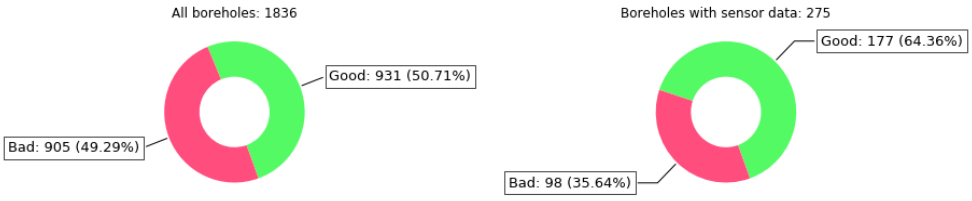

Figure 1: Ratio of good and bad boreholes after preprocess.

If we focus on the boreholes with data (since those boreholes are the input of our models), the number of sensor events is 41,218. Despite this number could seem very high, we cannot use all of them due to some events may be too short. Looking into the lengths of the sensor events, we find that the median is in 56 and 96.11% of events are 500 (or less) in length (see Figure 2).

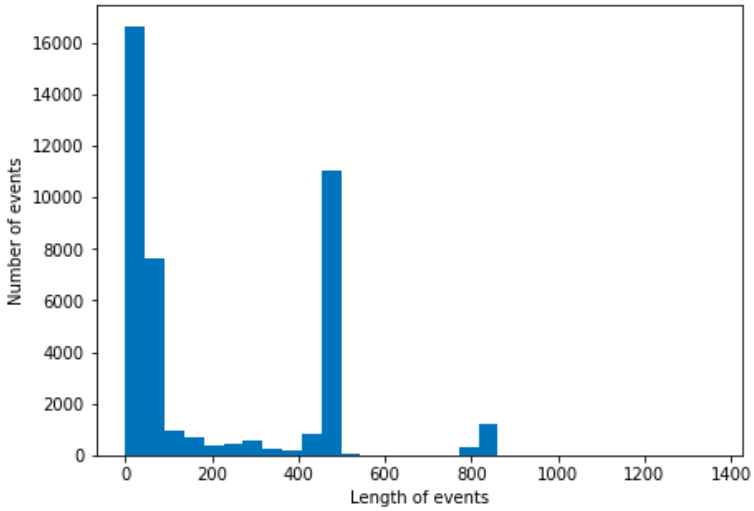

Figure 2: Lengths of the events.

In an effort to consider only representative time series as input of our models, we filter the events such that they must be at least 60 in length as we consider that they do not contain enough information. This filter results in 19,554 events (i.e.

only 47,44% of all events are larger than 60 values). Lastly, we need that all time series are in the same length, so we have to find the correct threshold to *cut* the long sequences and remove the short ones.

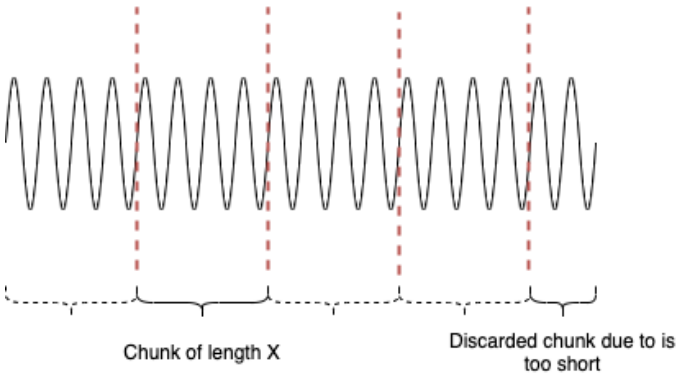

Figure 3: Example of a time series split into chunks of a fixed length.

We find the threshold value by executing a quick analysis (over all the dataset) to know how many samples per sensor there are depending on the threshold value (see Table 1). As a result of analysing different thresholds, we select 500 as the length of time series chunks. In other words, we split the time series into smaller chunks of length 500 and remove the chunks which are shorter than that (see Figure 3). Also, we add one more restriction: in order to train a model using the data of a sensor, we must have at least 6 samples of each class: 6 *good* samples and 6 *bad* ones.

In the case of LSTM, there is a common strategy for batch optimization: *zero-padding*. This implies that we do not need to remove the short time series (shorter than 500 values) but dilate them with zeros. This is specially useful when modeling temporal data where the model should not violate the temporal order [6]. However, we recall that if a time series chunk is shorter than 60, we remove it.

Secondly, the FD analysis looks much better than the previous one: all the time series are suitable to use as input of a model since all of them are sequences with 500 values. Regarding to the *good* and *bad* noise engines, the data is slightly unbalanced: from the 3601 measurements, there are 1755 *good* samples and 1846 *bad* ones (see Figure 4).

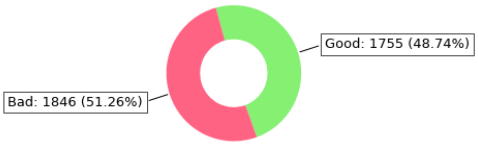

Figure 4: Ratio of good and bad noise measurements.

2.4 Combination of sensor data

Some models allow us to make predictions over multi-dimensional time series (see Figure 5). This is interesting if we want to use two (or more) time series from different sensors in the exactly same time windows, so we can study the correlation between them. For example, following the previous analysis, the input in the LSTM model would be $\texttt{batch size} \times 500 \times \#S$ where #S is the number of sensors that we are combining.

After evaluating the intersections of all time series in each different combination of sensors, we consider that is more precise to train a model for each sensor. In consequence, the input shape is $\texttt{batch size} \times 500 \times 1$ where *batch size* depends on

Time Series Chunks Length	200		500	
Class of sample	Pos.	Neg.	Pos.	Neg.
Sensor 1	40 (76)	29011 (29833)	2 (51)	11626 (12360)
Sensor 2	24 (74)	27519 (28340)	1 (56)	10766 (11711)
Sensor 3	40 (59)	25599 (26302)	7 (35)	9918 (10836)
Sensor 4	26 (55)	28145 (28985)	12 (37)	10978 (11952)
Sensor 5	56 (93)	24813 (25531)	14 (56)	9531 (10542)
Sensor 6	17 (53)	17602 (19334)	0 (42)	6246 (8421)
Sensor 7	67 (132)	32325 (33100)	15 (86)	12493 (13546)
Sensor 8	16 (25)	19966 (20888)	0 (14)	7812 (8798)
Sensor 9	18 (35)	26298 (27198)	3 (23)	10470 (11334)
Sensor 10	36 (95)	16618 (17439)	6 (70)	6390 (7404)
Sensor 11	45 (91)	16085 (16785)	7 (64)	5957 (7010)
Sensor 12	17 (78)	16289 (17150)	1 (68)	6205 (7191)
Sensor 13	15 (27)	31712 (32378)	0 (16)	12231 (13294)
Sensor 14	29 (82)	32908 (33681)	3 (62)	12737 (13758)
Sensor 15	84 (145)	32921 (33689)	21 (90)	12739 (13762)
Sensor 16	78 (124)	32911 (33682)	22 (74)	12692 (13738)
Sensor 17	84 (149)	32921 (33689)	21 (94)	12739 (13762)
Sensor 18	84 (149)	32921 (33689)	21 (94)	12739 (13762)
Sensor 19	84 (149)	32921 (33689)	21 (94)	12739 (13762)
Sensor 20	29 (81)	32888 (33627)	3 (61)	12690 (13711)
Sensor 21	34 (54)	30827 (31633)	4 (34)	12414 (13150)
Sensor 22	34 (61)	30687 (31561)	4 (41)	12414 (13150)
Sensor 23	84 (149)	32921 (33689)	21 (94)	12739 (13762)
Sensor 24	84 (149)	32921 (33689)	21 (94)	12739 (13762)
Sensor 25	84 (149)	32921 (33689)	21 (94)	12739 (13762)
Sensor 26	0 (9)	102 (178)	0 (9)	34 (110)
Sensor 28	3 (7)	800 (922)	0 (5)	0 (505)

Table 1: Summary of number of chunks per sensor using 200 and 500 as chunk size. In parenthesis, there is the number of samples in case that, instead of removing the short time series, we use zero-padding.

the size of the set of samples of each sensor. In short, the input of our models are univariate time series.

3 MODELS

3.1 LSTM

Long Short-Term Memory (aka. LSTM) is an artificial Recurrent Neural Network (aka. RNN) architecture. This kind of architectures are useful in problems of time series for the reason that they can use the feedback from a previous neuron as input of the next one.

Hochreiter, in his diploma thesis, explore the problem of RNN to learn long-term dependencies [3]. For example, in a language model trying to predict the next word based on the previous ones, it could predict the last word in the sentence "The cars are in the *road*" easily (it does not need any further context). However, sometimes it faces cases where it needs more context. It could happen that, to predict the last word, it requires information which appears further back[4]. Unfortunately, as that gap grows, RNNs become unable to learn to connect the information, and LSTM should be able to overcome this problem.

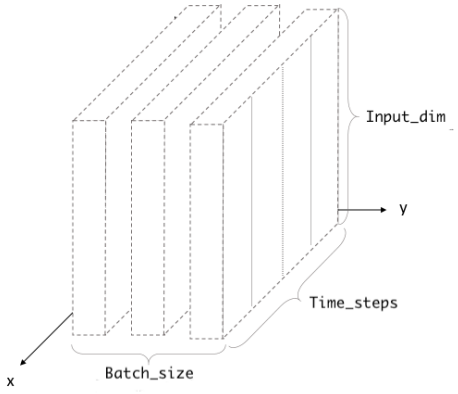

Figure 5: Shape of input with multi-dimensional time series.

Later, S. Hochreiter and J. Schmidhuber[1] explain that in common/vanilla RNN models "*error signals flowing backwards in time tend to either blow up or vanish*". In other words, when training a vanilla RNN using back-propagation, the gradients (which are back-propagated) can *vanish* (tend to zero) or *explode* (tend to infinity), because of the computations involved in the process, which use finite-precision numbers. In this sense, RNNs using LSTM can partially solve the vanishing gradient problem (but still they could *blow up*).

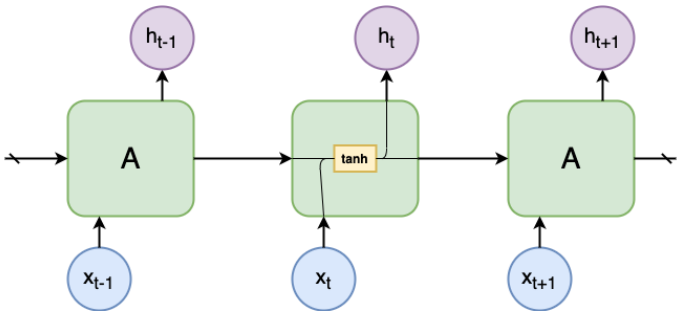

Figure 6: Architecture of a vanilla RNN.

S. Hochreiter and J. Schmidhuber propose to do some changes in the architecture of RNN models (see Figure 6). In short, they add three *regulators* (aka. gates) to the flow of information inside the LSTM unit: an input gate, an output gate and a forget gate (see Figure 7). Note that there are other variations of LSTM units which have different gates.

You can find some details about the implementation in the Appendix A.1.

3.2 Attention

Originally, the main goal of this type of models was to build a neural machine translator. The usual approach was based in an architecture encoder-decoder, but the performance deteriorated rapidly as the length of an input sentence increases.

There are different types of Attention[8] and, in this project, we adopt the approach described by Bahdanau et al.[8]. The new architecture consists of a Bidirectional RNN (BiRNN) as an encoder and a decoder that emulates searching through a source sentence during decoding a translation (see Figure 8). The idea is that, given an input sequence $x = (x_1, \ldots, x_{T_x})$, it embeds the sequence into a context vector c and then, based on this context vector, it produces the translation $y = (y_1, \ldots, y_{T_y})$.

8 Lilian Weng gives a detailed explanation at https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html

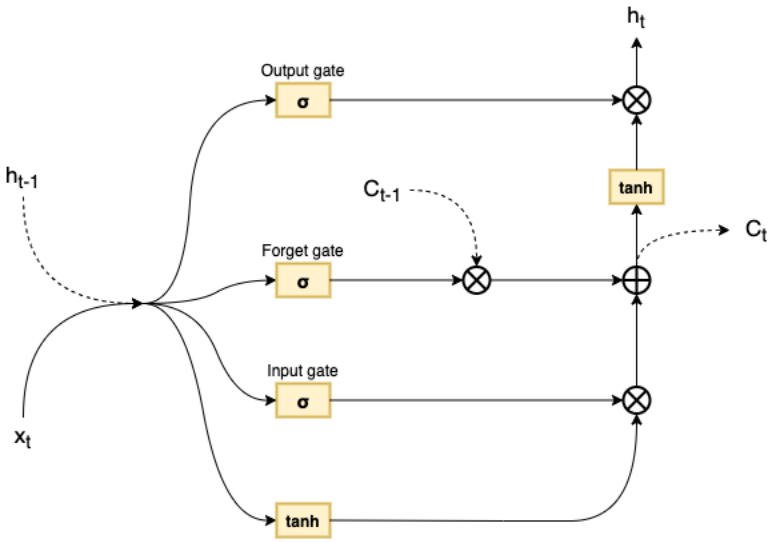

Figure 7: Architecture of a common LSTM unit.

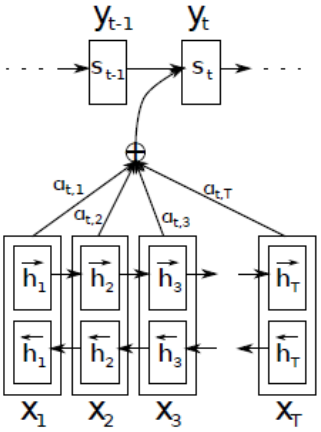

Figure 8: Graphical illustration proposed by Bahdanau et al. to generate the t-th target word y_t given a source sentence $x = (x_1, \ldots, x_{T_x})$.

A BiRNN consists in two RNN layers (in our case, we use LSTM) where the first layer reads the input sequence as it is ordered (from x_1 to x_{T_x}) and the second layer reads the input sequence in reverse order. As a result of these two layers, for each element x_j in the sequence x, we obtain two hidden states ($\overrightarrow{h_j}$ and $\overleftarrow{h_j}$) which we need to concatenate such that $h_j = \left[\overrightarrow{h_j}^\top ; \overleftarrow{h_j}^\top\right]$.

This context vector is computed as a weighted sum of the hidden states h_i:

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j$$

where α_{ij} are called *attention weights*. These weights are computed using an *alignment model* which scores how well the inputs around position j and the output at position i match:

$$\alpha_{t,i} = \text{align}(y_t, x_i) = \frac{\exp(\text{score}(s_{t-1}, h_i))}{\sum_{k=1}^{T_x} \exp(\text{score}(s_{t-1}, h_k))}$$

The score function is the non-linear function $\text{score}(s_t, h_i) = \mathbf{v}^\top \tanh(\mathbf{W}[s_t; h_i])$ where both $\mathbf{v}$ and $\mathbf{W}$ are weight matrices to be learned in the alignment model.

Recall that our problem is a Binary TCS, thus the equations of the model are simpler. In short, it implies that the index t in the previous equations is always 1 as we have one only output (see Figure 9).

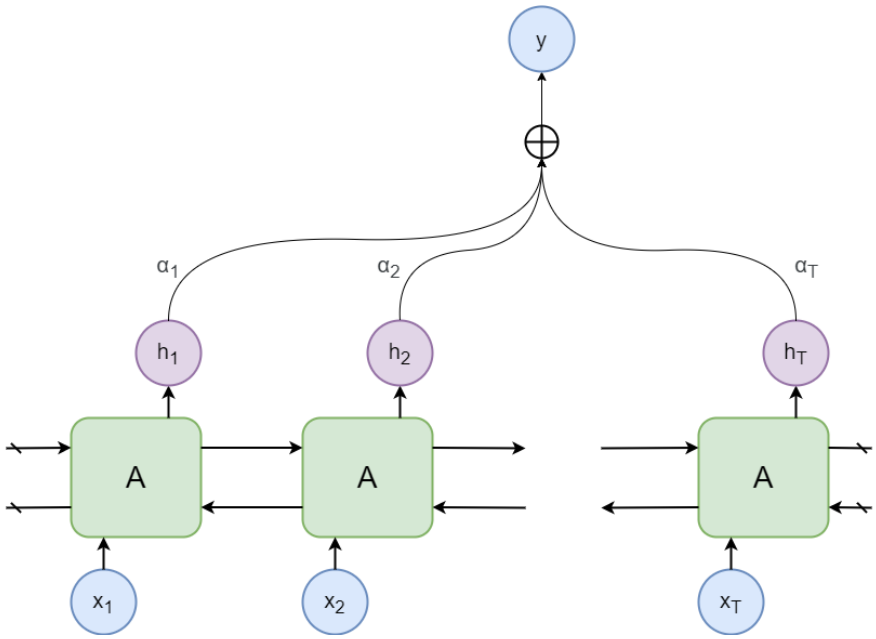

Figure 9: Simplified illustration of the Binary TSC problem using the approach defined by Bahdanau et al.

You can find some details about the implementation in the Appendix A.1.

4 EVALUATION

4.1 LSTM

In the case of BD, the predictions given by the LSTM model may seem quite good according to the accuracy. However, since the data is extremely unbalanced, we should look into the F1 Scores and the results are really bad (see Table 2). These bad results might be caused by the bad quality of the *Boreholes Dataset*.

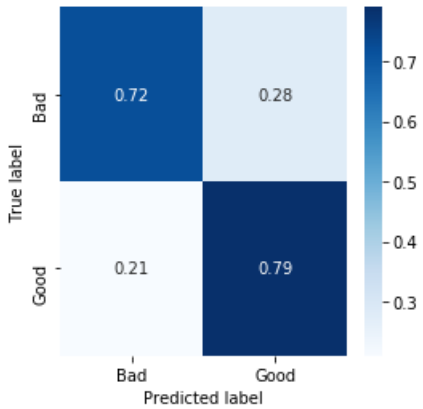

Figure 10: Confussion matrix of the predictions of LSTM in the *Ford Dataset*.

Nonetheless, when we train the same model with FD, it gives much better results (see Figure 10): the F1 score is 0.64 and the accuracy is 0.75. Though the quality

	TP	FP	TN	FN	Accuracy	F1 Score
Sensor 1	0.01	0.99	0.99	0.01	0.9883	0.1129
Sensor 2	0.01	0.99	0.99	0.01	0.9879	0.0388
Sensor 3	0.04	0.96	0.96	0.04	0.9542	0.0480
Sensor 4	0.04	0.96	0.96	0.04	0.9617	0.0222
Sensor 5	0.08	0.92	0.92	0.08	0.9154	0.0189
Sensor 6	0.99	0.01	0.01	0.99	0.0149	0.0121
Sensor 7	0.00	1.00	1.00	0.00	0.9963	0.0113
Sensor 10	0.02	0.98	0.98	0.02	0.9842	0.0144
Sensor 11	0.02	0.98	0.98	0.02	0.9791	0.0217
Sensor 12	0.01	0.99	0.99	0.01	0.9892	0.0213
Sensor 15	0.08	0.92	0.92	0.08	0.9224	0.0154
Sensor 17	0.91	0.09	0.09	0.91	0.0999	0.0139
Sensor 18	1.00	0.00	0.00	1.00	0.0075	0.0149
Sensor 19	0.01	0.99	0.99	0.01	0.9842	0.0183
Sensor 20	0.07	0.93	0.93	0.07	0.9329	0.0126
Sensor 22	0.05	0.95	0.95	0.05	0.9448	0.0632
Sensor 23	0.26	0.74	0.74	0.26	0.7447	0.0140
Sensor 24	0.02	0.98	0.98	0.02	0.9801	0.0125
Sensor 26	0.41	0.59	0.59	0.41	0.6111	0.2222

Table 2: Results of the LSTM model for each sensor. There are only displayed the sensors where the F1-Score of the model is greater than 0.01. TP: True positives, FP: False positives, TN: True negatives, FN: False negatives.

of BD and FD is not the same, we observe that it is possible to obtain good results using LSTM.

4.2 Attention

Similarly to the previous model with the BD, the Attention model returns bad results regarding to the F1 Score (see Table 3), not to mention that many sensors dissapear from the results (in comparison to the LSTM model) if we filter the models where the F1 Score is greater than 0.01.

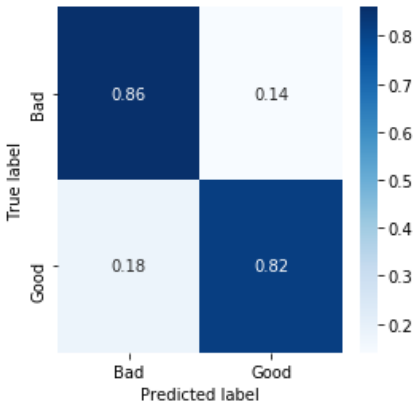

Figure 11: Confussion matrix of the predictions of Attention in the *Ford Dataset*.

However, the Attention model trained with the FD performs slightly better than the LSTM model (see Figure 11): the accuracy is 0.84 and the F1 Score is 0.67.

	TP	FP	TN	FN	Accuracy	F1 Score
Sensor 6	0.99	0.01	0.01	0.99	0.0161	0.0104
Sensor 7	0.00	1.00	1.00	0.00	0.9958	0.0137
Sensor 10	0.01	0.99	0.99	0.01	0.9846	0.0179
Sensor 11	0.04	0.96	0.96	0.04	0.9629	0.0170
Sensor 12	0.01	0.99	0.99	0.01	0.9878	0.0185
Sensor 16	0.06	0.94	0.94	0.06	0.9388	0.0116
Sensor 17	0.05	0.95	0.95	0.05	0.9452	0.0144
Sensor 18	0.07	0.93	0.93	0.07	0.9275	0.0125
Sensor 19	0.08	0.92	0.92	0.08	0.9217	0.0135
Sensor 23	0.05	0.95	0.95	0.05	0.9447	0.0140
Sensor 24	0.08	0.92	0.92	0.08	0.9227	0.0106
Sensor 25	0.07	0.93	0.93	0.07	0.9319	0.0135
Sensor 26	0.44	0.56	0.56	0.44	0.5833	0.1053

Table 3: Results of the Attention model for each sensor. There are only displayed the sensors where the F1-Score of the model is greater than 0.01. TP: True positives, FP: False positives, TN: True negatives, FN: False negatives.

5 CONCLUSIONS

We presented two different approaches in how to apply deep learning for Time Series Classification. In particular, we described two models where we use LSTM to do a binary classification of time series.

RNN-based models are a good choice when we need that the information persists during the training. However, long sequences may cause the well-known problem *vanishing gradient*. LSTM avoid this problem, although it does not solve another problem of RNN: *exploding gradient*. Furthermore, we require 4 linear layers per cell, which increases memory-bandwidth-bound computation.

We showed two mechanishms to avoid another problem of LSTM: overfitting. The idea to solve the problem is quite different in Dropout and Attention, but both approaches cause that LSTM learns where to look for the relevant information.

Although both Dropout and Attention gave us good results when we used the *Ford Dataset* as input of the models, it was just the opposite when we talk about the *Boreholes Dataset*. In my opinion, this fact evidences that the quality of *Boreholes Dataset* is not good enough to make predictions about it. I believe that, if we fill the gaps in the data of sensors with new data, we will be able to achieve better results, or even use not only the time series but also other features as input of the model.

REFERENCES

[1] S. Hochreiter and J. Schmidhuber, *Long Short-Term Memory*. Neural Computation, 9 (8): p1735–1780, 1997. doi:10.1162/neco.1997.9.8.1735.

[2] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho and Yoshua Bengio, *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling*. arXiv:1412.3555 [cs.NE], 2014.

[3] S. Hochreiter, *Untersuchungen zu dynamischen neuronalen Netzen* [German]. Diploma thesis, Institut f. Informatik, Technische Univ. Munich, 1991.

[4] Pranjal Srivastava, *Essentials of Deep Learning: Introduction to Long Short Term Memory* (Dec. 2017). Published at `https://www.analyticsvidhya.com/blog/2017/12/fundamentals-of-deep-learning-introduction-to-lstm/` [Accessed on 27-04-2020]

[5] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever and Ruslan Salakhutdinov, *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*. Journal of Machine Learning Research 15, 1929-1958, 2014.

[6] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior and Koray Kavukcuoglu, *WaveNet: A Generative Model for Raw Audio* (2016).

[7] Hassan Ismail Fawaz, Germain Forestier, Jonathan Weber, Lhassane Idoumghar and Pierre-Alain Muller, *Deep Learning for Time Series Classification: a Review*, arXiv:1809.04356 [cs.LG], 2019.

[8] Dzmitry Bahdanau, Kyunghyun Cho and Yoshua Bengio, *Neural Machine Translation by Jointly Learning to Align and Translate*. arXiv:1409.0473 [cs.CL], 2014.

A APPENDIX

A.1 Models in code

A.1.1 *LSTM*

We use Keras[9] (running on top of TensorFlow), which already has an implementation of LSTM, to create our model:

```
model = Sequential([
LSTM(100,
input_shape=(sequence_length, 1),
dropout=0.3,
recurrent_dropout=0.3,
return_sequences=True),
Flatten(),
Dense(100, activation='relu'),
Dense(1, activation='sigmoid')
])
model.compile(optimizer=tf.keras.optimizers.Adam(),
loss=tf.keras.losses.BinaryCrossentropy(),
metrics=...)
```

Since LSTM tends to overfit, there is a *Dropout* layer [5] after each *LSTM* layer. As it name says, these layers drop out units (both hidden and visible) in the neural network to prevent overfitting. Basically, it randomly set a fraction of input units to 0 at each update during training time (in our case, the fraction rate is 0.2).

At the end, we add a *Flatten* layer and, later, a *Dense* layer to reduce the dimension to 1 only output, that is, the prediction. Note that we use *sigmoid* as activation function and *Binary Cross-entropy* as loss function due to this problem is a binary classification.

Moreover, we should not forget the fact that selection of hyperparameters values is important to get the best posible prediction. This is out of the scope of this project, so you can find the details of the optimization process at Optimization of hyperparameters.

Before we proceed to train the model, we should recall that the BD is very unbalanced. For this reason, it is necessary to adjust the weights of the classes before we can start the training. Hopefully, Scikit-Learn already contains a function[10] to calculate the weights:

```
weights = class_weight.compute_class_weight('balanced',
np.unique(Y_train), Y_train)
weights = dict(enumerate(weights))

model.fit(
X_train,
Y_train,
epochs=7,
validation_data=(X_test, Y_test),
class_weight=weights,
verbose=1
)
```

9 https://keras.io/

10 https://scikit-learn.org/stable/modules/generated/sklearn.utils.class_weight.compute_class_weight.html

A.1.2 Attention

Before we proceed to create our model, we define the Attention layer that we are going to use later:

```
class Attention(tf.keras.layers.Layer):
def __init__(self, units=16):
super(Attention, self).__init__()
self.W1 = tf.keras.layers.Dense(units)
self.W2 = tf.keras.layers.Dense(units)
self.V = tf.keras.layers.Dense(1)

def call(self, features, hidden):
hidden_with_time = tf.expand_dims(hidden, 1)
score = tf.nn.tanh(self.W1(features) +
self.W2(hidden_with_time))
attention_weights = tf.nn.softmax(self.V(score), axis=1)

context_vector = attention_weights * features
context_vector = tf.reduce_sum(context_vector, axis=1)

return context_vector, attention_weights
```

Now that we have all the pieces to build the model, the first step is to define the input of the model, which is pretty much the same as the LSTM model:

```
input_layer = Input(shape=(input_length, 1), name='encoder_inputs')
```

As it is explained above, we need a BiRNN in the model. In our case, we use LSTM as we already explained how it works.

```
hidden_size = 64
lstm_layer, forward_h, forward_c, backward_h, backward_c = Bidirectional(
LSTM(hidden_size,
dropout=0.3,
return_sequences=True,
return_state=True), name='bi_lstm')(input_layer)
```

Now we are in the step where we need to concatenate the hidden states and use them to build the Attention layer.

Note that we obtain not only the output of the BiRNN but also the hidden states. It is necessary to concatenate the hidden states of BiRNN in order to add the Attention layer.

```
state_h = Concatenate()([forward_h, backward_h])
context_vector, attention_weights = Attention()(lstm_layer, state_h)
```

Finally, we add a fully-connected layer and the layer which is going to give us the prediction. Furthermore, we take the same optimizer and loss functions as in the LSTM model.

```
dense_layer = Dense(32, activation='relu')(context_vector)
output_layer = Dense(1, activation='sigmoid')(context_vector)

model = tf.keras.Model(inputs=input_layer, outputs=output_layer)

model.compile(optimizer=tf.keras.optimizers.Adam(),
loss=tf.keras.losses.BinaryCrossentropy(),
metrics=...)
```

A.2 Optimization of hyperparameters

The hyperparameters are all the parameters which can be arbitrarily set by the user before starting the training. In this project, we use Tensorflow to build and train the different models. In this library, there is not exist any built-in function to optimize the hyperparameters. Therefore, we should implement a simple function to compute all the permutations of parameters that we want to test:

```
# Recursive function to find all combinations of parameters
def tunning_params_combinations(tunning_params, combination={}):
    if len(tunning_params) == 0:
        return [combination]

    c_combinations = []

    # Get first param
    param_name = next(iter(tunning_params))
    param_values = tunning_params[param_name]

    for v in param_values:
        c = copy.deepcopy(combination)
        sub_tp = copy.deepcopy(tunning_params)

        del sub_tp[param_name]
        c[param_name] = v

        c_combinations = c_combinations + tunning_params_combinations(sub_tp, c)

    return c_combinations

# Define parameters that we want to use in the training
tunning_params = {
    'lstm_activation': ['tanh', 'redu'],
    'dropout_1': [0.2, 0.4],
    'dropout_2': [0.2, 0.4]
}

# Get combinations
param_combinations = tunning_params_combinations(tunning_params)

# Use every item of "param_combinations" to create
# a LSTM model and train it
```

It is important to keep in mind that the mentioned method compute all possible permutations of parameters, and this directly affects the total training time. Moreover, we might not find the best hyperparameter values using this method as it is very likely that we did not write the optimal values among the hyperparameters to use.