DETECTING EVENTS IN TEMPORAL LOG DATA

National Research University Higher School of Economics
Moscow, Russia

DIMAS MUNOZ MONTESINOS
SUPERVISOR: ILYA MAKAROV

May 26th, 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 depend more on automated 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. In this project, we show two approaches of using Neural Networks which are successful in this problem: LSTM and Attention models.

CONTENTS

1 INTRODUCTION

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

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, it is used for signal detection and estimation; and, in the context of pattern recognition, it may be used for clustering or anomaly detection. It is good to point out that TSC differs from common classification problems since the data is ordered in sequences. In this project, we try to apply an algorithm which allows us to split the time series into two categories, that is, a binary classification.

There are many algorithms out there which promise us to classify the data more or less accurate. Particularly, the TSC tasks were commonly solved by using Recurrent Neural Networks (RNNs). 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 classification problems, such as LSTM, GRU or SVM to mention a few ones.

Before we continue, we use some terms in the next sections which we should have in mind to fully understand the explanations:

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 of 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 these definitions, we take the definition of TSC given by Ismail Fawaz et al. to write more formally the problem: *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)* [1].

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, except the dimension (which it depends on whether we have univariate or multivariate time series).

We cover two different approaches to arrange the data: LSTM and Attention. As we explain deeper in the next sections, both of them are well-known in the field of time series forecasting and classification. In order to implement them, we use different technologies:

- Python 3.7 to preprocess the data and train the model(s). Along with it, we use many third-party libraries, such as Tensorflow[1] and Keras[2] among others.
- 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 https://www.tensorflow.org/api_docs/python/tf
2 https://keras.io/

2 DATASETS

2.1 Description

To explore the algorithms and their accuracy, we use two different datasets of time series. The first dataset is based on the data collected by multiple sensors during the excavation of boreholes. The second dataset consists of engine noise meauserements from vehicles of Ford brand.

The first dataset (which we are going to call *Boreholes Dataset* or *BD*), as we see in the further sections, is not "ideally" collected: there are many issues and gaps that we must fix before we can use the data. It is very likely that we cannot obtain good results from such dataset, so we also expertiment with the second dataset[3] (which we will call *Ford Dataset* or *FD*). Unlike the BD, the FD has been collected in optimal conditions, hence we should obtain reliable conclusions about the performance of the algorithms.

Respecting to the content of the datasets, on the one hand, the BD consists of 11 GB of data of exported SQL data from Postgresql[4] and 2.4 MB of data divided in several CSV files. The data data collected by each sensor in the time is grouped in each SQL file: position of the driller, pressure, temperature, spindle speed... The CSV files contain information about the properties and parameters of sensors and boreholes:

- The properties of the sensors: sensor name and the units of time series.
- The experiments with the starting and finish times. Each experiment consists of 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 dataset was 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 its content in "usable" data.

2.2 Preprocessing

2.2.1 *Boreholes*

To be able to locate the sensor data of a borehole, we must know the start and stop time of the boreholes. Nevertheless, there are some of them in the dataset which do not have neither start/stop time nor any information to infer the time. Since we cannot locate these boreholes in the time, we have removed them from the "cleaned dataset" as we cannot find their sensor data either.

2.2.2 *Phases of boreholes*

Each borehole is composed of some phases: a borehole may contain zero, one or even 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.

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

4 https://www.postgresql.org/

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 infer the empty values by inheriting the value(s) of the phase in the very previous time step.

During this process, we face one more problem: some phases do not contain information about the time. Similarly to the boreholes without time information, we cannot sort these phases by time, thus it is not possible to inherit the value of the previous one. In such case, we do not remove them but skip them.

2.2.3 *Sensor data*

Each sensor include long numerical sequences with some "noise". We clean the data of each one independently:

- Remove all 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). To illustrate it with an example, if there are 4 values and, one minute later, there are 10 values, then we remove the first 4 values.
- Take the time windows of all boreholes (that is, an interval of time) and remove the sensor data which is not present in the union of those time windows.
- The sensors are collecting many values per second and it likely comes up with "peaks" in little intervals. In a naive way to simplify the large time series and reduce those peaks, we reduce all values (in intervals of one second) to the average of them.

Clearly, we see that the data 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 $x \in \mathbb{R}_+$ and Y_i is the label of the borehole. Also, informally writing, 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 lengths as each sensor is independent to the others.

2.2.4 *Sensor events*

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

Definition 4. A sensor event E_s (in a sensor s) is described as the maximal time interval 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:

$$x_k, x_{k+1} \in E_s \Leftrightarrow d(x_k, x_{k+1}) \leqslant \alpha$$

In our experiments with the data 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 consecutive 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 others which do not contain any sensor data at all (thus they do not have any sensor event).

2.2.5 *Classes of boreholes*

Based on some criteria, we assign a label 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*.

Equivalently, 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 empty values are correct. For example, if the only value we know is that *roundness mean* is less than 0.06, then we assume that the borehole is *good*.

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 are 44.63% more good boreholes with sensor data than bad ones.

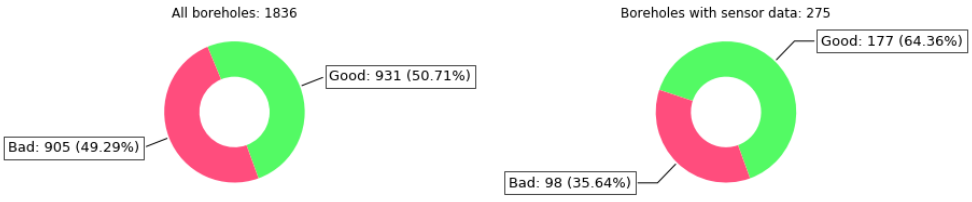

Figure 1: Ratio of good and bad boreholes after preprocess.

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 out that the median is 56 and 96.11% of events are 500 (or less) in length (see Figure 2).

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

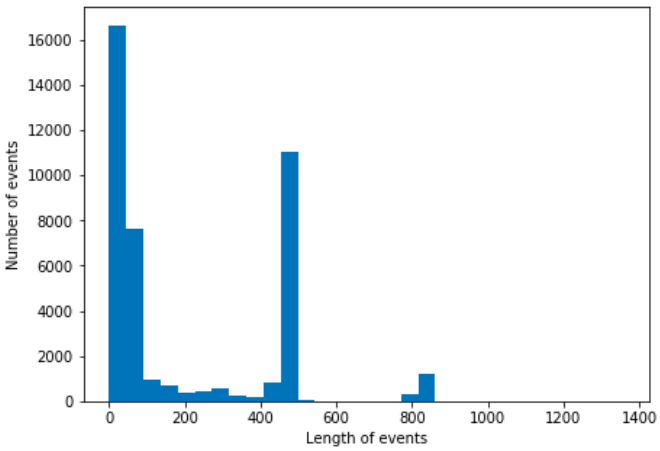

Figure 2: Lengths of the events.

series are in the same length, so we require an appropiate threshold to *cut* the long sequences and remove the short ones.

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 single sensor, we must have at least 6 samples of each class: 6 *good* samples and 6 *bad* ones.

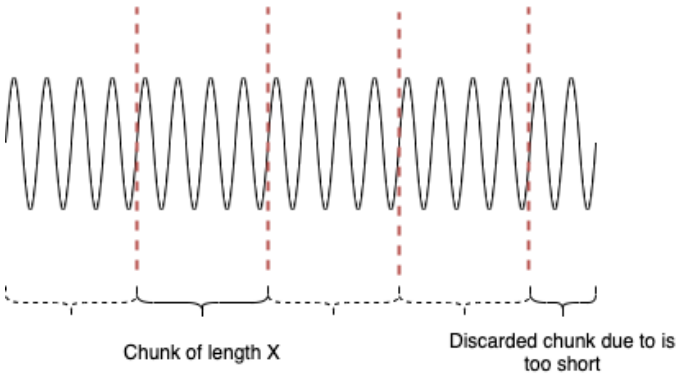

Figure 3: Abstract example of a time series split into chunks of a fixed length.

In the particular case of LSTM, there is a common strategy for batch optimization called *zero-padding*. This strategy 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 [2]. 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 be used as input of a model since all of them are sequences with 500 values. Regarding to the *good* and *bad* labels of noise engines, the data is slightly unbalanced: from the 3601 measurements, there are 1755 *good* samples and 1846 *bad* ones (see Figure 4).

Time Series Chunks Length	200		500	
Class of sample	Pos.	Neg.	Pos.	Neg.
Feed	40 (76)	29011 (29833)	2 (51)	11626 (12360)
ForceW1	24 (74)	27519 (28340)	1 (56)	10766 (11711)
ForceW2	40 (59)	25599 (26302)	7 (35)	9918 (10836)
ForceW3	26 (55)	28145 (28985)	12 (37)	10978 (11952)
ForceW4	56 (93)	24813 (25531)	14 (56)	9531 (10542)
LtiCurX1	17 (53)	17602 (19334)	0 (42)	6246 (8421)
LtiCurY1	67 (132)	32325 (33100)	15 (86)	12493 (13546)
LtiCurZ	16 (25)	19966 (20888)	0 (14)	7812 (8798)
LtiInPar1	18 (35)	26298 (27198)	3 (23)	10470 (11334)
LtiPosX1	36 (95)	16618 (17439)	6 (70)	6390 (7404)
LtiPosY1	45 (91)	16085 (16785)	7 (64)	5957 (7010)
LtiPosZ	17 (78)	16289 (17150)	1 (68)	6205 (7191)
MqlFlow	15 (27)	31712 (32378)	0 (16)	12231 (13294)
MqlPressure	29 (82)	32908 (33681)	3 (62)	12737 (13758)
NcState	84 (145)	32921 (33689)	21 (90)	12739 (13762)
Override	78 (124)	32911 (33682)	22 (74)	12692 (13738)
PosX	84 (149)	32921 (33689)	21 (94)	12739 (13762)
PosY	84 (149)	32921 (33689)	21 (94)	12739 (13762)
PosZ	84 (149)	32921 (33689)	21 (94)	12739 (13762)
RowId	29 (81)	32888 (33627)	3 (61)	12690 (13711)
SpindleRpm	34 (54)	30827 (31633)	4 (34)	12414 (13150)
SpindleSetRpm	34 (61)	30687 (31561)	4 (41)	12414 (13150)
TempAmbient	84 (149)	32921 (33689)	21 (94)	12739 (13762)
TempControlCabinet	84 (149)	32921 (33689)	21 (94)	12739 (13762)
TempWorkingArea	84 (149)	32921 (33689)	21 (94)	12739 (13762)
VelX	0 (9)	102 (178)	0 (9)	34 (110)
VelY	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.

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 shape in the LSTM model could be $\texttt{batch size} \times 500 \times \#S$ where #S is the number of sensors that we are combining.

We evaluate the intersections of time series of different sensors, seeking for a combination of sensors where the data may be suitable to be used as a multivariate time series. Unfortunately, the intersections are either very short or there are very few ones, thus we consider that it is more precise to train a model which the data of each sensor independently (in short, the input of the models are univariate time series). In consequence, the input shape is $\texttt{batch size} \times 500 \times 1$ where *batch size* depends on the size of the set of samples of each sensor.

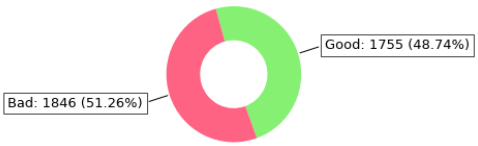

Figure 4: Ratio of good and bad noise measurements.

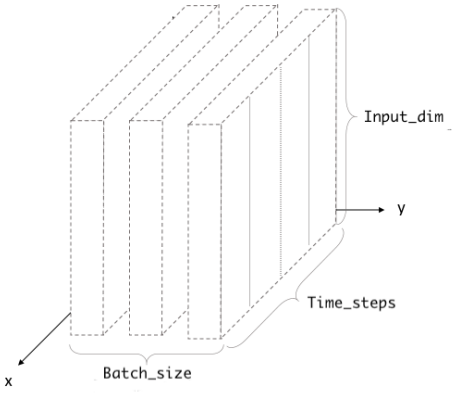

Figure 5: Shape of input with multi-dimensional time series.

3 MODELS

3.1 LSTM

Long Short-Term Memory (more known as LSTM) is an artificial Recurrent Neural Network (RNN) architecture capable of learning long-term dependencies. 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.

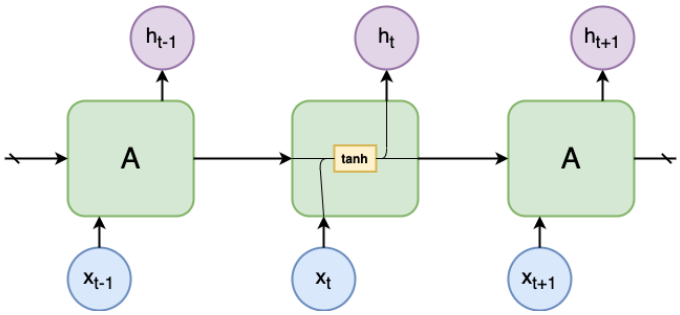

Figure 6: Architecture of a vanilla RNN.

Hochreiter, in his diploma thesis, explored the problem of RNN to learn long-term dependencies [3]. For example, a language model which is trying to predict the next word based on the previous ones, it could suggest 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, common RNNs become unable to learn to connect the information and LSTM should be able to overcome this problem.

Later, S. Hochreiter and J. Schmidhuber explained that in common/vanilla RNN models "*error signals flowing backwards in time tend to either blow up or vanish*" [5]. 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*).

Briefly explained, S. Hochreiter and J. Schmidhuber propose to do some changes in the architecture of RNN models (see Figure 6). They add three *regulators* (also known as *gates*) to the flow[5] of information inside the LSTM unit: an input gate, an output gate and a forget gate (see Figure 7). LSTM keeps the *chain-like* structure of vanilla RNNs but, instead of having one only neural network layer per cell, they have four of them now. It is important to keep in mind this fact due to LSTM demands more resources.

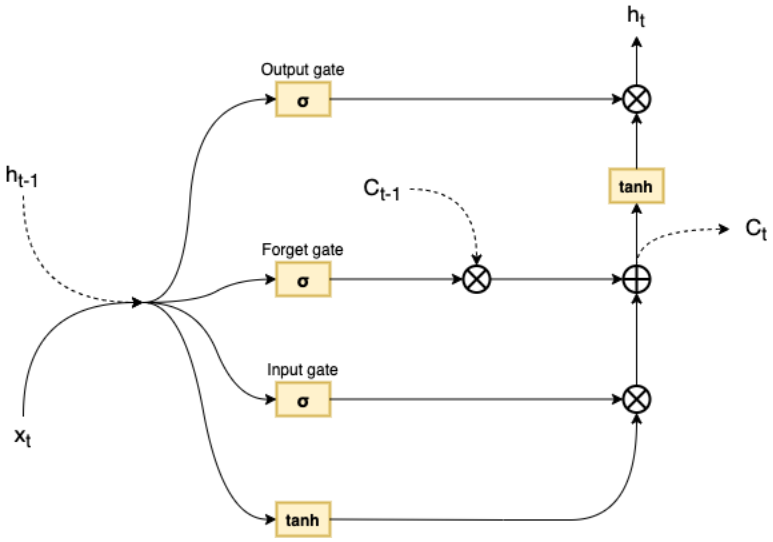

Figure 7: Architecture of a common LSTM unit.

It is good to remark that there are other variations of LSTM units which have different gates. For example, one famous variation is GRU (introduced by Cho et al. [6]).

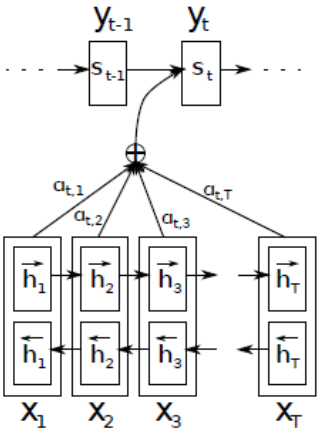

Figure 8: Graphical illustration proposed by Bahdanau et al. to generate the t-th target word y_t given a source sentence $x = (x_1, \ldots, x_{T_x})$.

LSTM suffers from one more problem: *overfitting*. In this case, the algorithm will be considering as valid only the data identical to that of the training set and being unable to distinguish good and reliable inputs if they go a little outside the

5 Christopher Olah explains the gates as well as LSTM variations in his blog: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

already established ranges. For such reason, we must use LSTM along a technique to avoid overfitting: Dropout. This new layer was introduced by N. Srivastava et al. in 2014[7]. In a nutshell, Dropout, as its names indicates, detaches off a certain percentage of information units, thus it makes possible to generalize information.

3.2 Attention

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

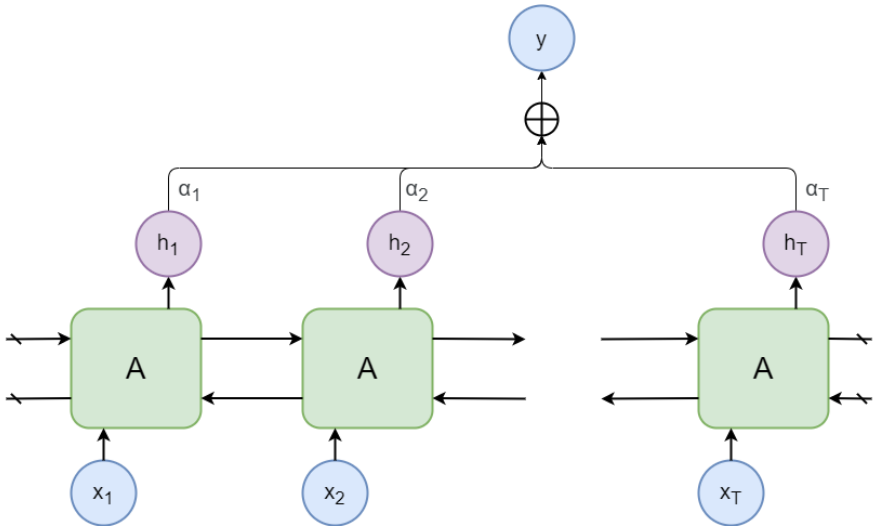

Figure 9: Simplified illustration the model Attention defined by Bahdanau et al. applied to the binary Time Series Classification problem.

There are different types of Attention[6] and, in this project, we adopt the approach described by Bahdanau et al. [8]. The new architecture consists of a Bidirectional RNN (BiRNN to abbreviate) as an encoder and a decoder that emulates searching through a source text during the decode process of a translation (see Figure 8). The main 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 $y = (y_1, \ldots, y_{T_y})$.

The BiRNN layer actually consists of two RNN layers 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 $h_j = \left[\overrightarrow{h_j}^{\top}; \overleftarrow{h_j}^{\top}\right]$. In our particular case, we use LSTM in the BiRNN layer as we want to know whether Attention improves the performance of the previous model (based in "simple" LSTM layers) or not.

Regarding to the context vector, it is computed as a weighted sum of the hidden states h_i

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

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

where α_{ij} are called *attention weights*. The 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).

4 EVALUATION

4.1 LSTM

In the case of the BD, the predictions that we obtain with 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 very poor (see Table 2). These bad results might be caused by the bad quality of the *Boreholes Dataset*.

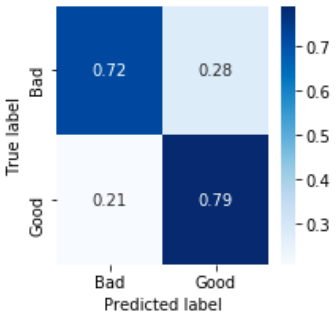

Figure 10: Confussion matrix of the predictions of LSTM (with Dropout) in the *Ford Dataset*.

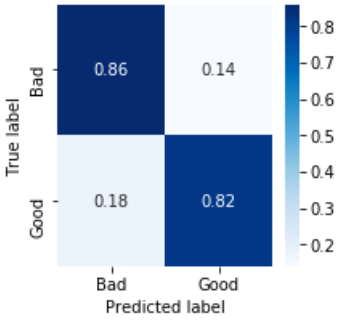

Figure 11: Confussion matrix of the predictions of Attention in the *Ford Dataset*.

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 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 respecting 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.

When we move to the Ford Dataset and we train the Attention model, it performs slightly better than the LSTM model (see Figure 11): the accuracy and the F1-Score grow up to 0.84 and 0.67 respectively.

	TP	FP	TN	FN	Accuracy	F1 Score
Feed	0.01	0.99	0.99	0.01	0.9883	0.1129
ForceW1	0.01	0.99	0.99	0.01	0.9879	0.0388
ForceW2	0.04	0.96	0.96	0.04	0.9542	0.0480
ForceW3	0.04	0.96	0.96	0.04	0.9617	0.0222
ForceW4	0.08	0.92	0.92	0.08	0.9154	0.0189
LtiCurX1	0.99	0.01	0.01	0.99	0.0149	0.0121
LtiCurY1	0.00	1.00	1.00	0.00	0.9963	0.0113
LtiPosX1	0.02	0.98	0.98	0.02	0.9842	0.0144
LtiPosY1	0.02	0.98	0.98	0.02	0.9791	0.0217
LtiPosZ	0.01	0.99	0.99	0.01	0.9892	0.0213
NcState	0.08	0.92	0.92	0.08	0.9224	0.0154
PosX	0.91	0.09	0.09	0.91	0.0999	0.0139
PosY	1.00	0.00	0.00	1.00	0.0075	0.0149
PosZ	0.01	0.99	0.99	0.01	0.9842	0.0183
RowId	0.07	0.93	0.93	0.07	0.9329	0.0126
SpindleSetRpm	0.05	0.95	0.95	0.05	0.9448	0.0632
TempAmbient	0.26	0.74	0.74	0.26	0.7447	0.0140
TempControlCabinet	0.02	0.98	0.98	0.02	0.9801	0.0125
VelX	0.41	0.59	0.59	0.41	0.6111	0.2222

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

	TP	FP	TN	FN	Accuracy	F1 Score
LtiCurX1	0.99	0.01	0.01	0.99	0.0161	0.0104
LtiCurY1	0.00	1.00	1.00	0.00	0.9958	0.0137
LtiPosX1	0.01	0.99	0.99	0.01	0.9846	0.0179
LtiPosY1	0.04	0.96	0.96	0.04	0.9629	0.0170
LtiPosZ	0.01	0.99	0.99	0.01	0.9878	0.0185
Override	0.06	0.94	0.94	0.06	0.9388	0.0116
PosX	0.05	0.95	0.95	0.05	0.9452	0.0144
PosY	0.07	0.93	0.93	0.07	0.9275	0.0125
PosZ	0.08	0.92	0.92	0.08	0.9217	0.0135
TempAmbient	0.05	0.95	0.95	0.05	0.9447	0.0140
TempControlCabinet	0.08	0.92	0.92	0.08	0.9227	0.0106
TempWorkingArea	0.07	0.93	0.93	0.07	0.9319	0.0135
VelX	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 showed up two different approaches in how to apply deep learning for Time Series Classification. Both of them are architectures which use LSTM to provide a solution to binary classification of time series.

In general, 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*. In consequence, we chose LSTM to avoid this problem, although it does not solve the second problem of RNN: *exploding gradient*. At the time of experimenting and finding the optimal configuration, we encountered this issue with certain hyperparameters. Furthermore, compared to vanilla RNN models, we required 4 linear layers per LSTM cell, which increased drastically the memory-bandwidth-bound computation.

Along with LSTM, we showed two mechanisms to avoid overfitting: one is embeeded in Attention and the other is the Dropout layer. The idea to solve the issue is quite different in Dropout and Attention, but both approaches cause that LSTM learns where to look for the relevant information. Not to mention that Attention performs better since it does not detaches off random information units (like Dropout) but gives a greater or lower weights to them.

Although both models 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. It may be possible that the time series in the dataset are very specific cases, so the model faces the opposite issue to overfit: underfit.

We have seen many gaps and short sequences, so I believe that, if we fill those gaps in the data of sensors with new data, we will be able to achieve better results using the *Boreholes Dataset*. Or even use not only the time series but also other features as input of the model.

REFERENCES

[1] Hassan Ismail Fawaz, Germain Forestier, Jonathan Weber, Lhassane Idoumghar, and Pierre-Alain Muller. Deep learning for time series classification: a review. *Data Mining and Knowledge Discovery*, 33(4):917–963, Mar 2019.

[2] 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.

[3] S. Hochreiter. Untersuchungen zu dynamischen neuronalen netzen, 1991. [German].

[4] Pranjal Srivastava. Long short term memory: Architecture of lstm, Dec 2017.

[5] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[6] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014.

[7] 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, 06 2014.

[8] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2014.