

UNIVERSITÃL' DE LORRAINE IDMC LORIA

MSc Natural Language Processing – 2018 - 2019
UE 805 – Supervised Project

# Anomaly detection with deep learning models
# Realisation report


*Students:*
Esteban Marquer
Prerak Srivastava

*Supervisors:*
Christophe Cerisara
Samuel Cruz-Lara
*Reviewer:*
Denis Jouvet


May 28, 2019

# Contents

**Introduction** **1**

**1 Datasets and preprocessing** **2**
1.1 BULL-ATOS dataset . . . 2
1.1.1 Pre-processing . . . 2
1.1.2 Usage . . . 2
1.2 Los Alamos National Laboratory (LANL) dataset . . . 2
1.2.1 Source of the data . . . 2
1.2.2 Structure of the log lines . . . 3
1.2.3 Pre-processing . . . 3
1.2.4 Usage . . . 4
1.2.4.1 *Percentile* corpus . . . 4
1.2.4.2 *Day 7* and *day 8* corpus . . . 4
1.3 BAREM dataset . . . 5
1.3.1 Source of the data . . . 5
1.3.2 Structure of the log lines . . . 5
1.3.2.1 General structure of the log lines . . . 7
1.3.2.2 Analysis of the dataset and descriptive statistics . . . 7
1.3.3 Pre-processing . . . 11
1.3.3.1 Calculating delta timestamps . . . 12
1.3.4 Usage . . . 12

**2 Predictive deep-learning models** **13**
2.1 General informations on our models . . . 13
2.1.1 Predictive event models . . . 13
2.1.2 Character embeddings . . . 13
2.1.3 Loss and model training . . . 14
2.1.3.1 Cross entropy loss and model output . . . 14
2.2 Deep Averaging Network (DAN)-based event model . . . 15
2.2.1 Predictive DAN . . . 15
2.2.2 Performance on the BULL-ATOS dataset . . . 16
2.2.3 Adaptation to the LANL dataset . . . 16
2.2.4 Attempts to achieve better accuracy and to avoid under-fitting . . . 16
2.2.4.1 Model with increased parameters . . . 17
2.2.4.2 Multi-line model with attention over lines . . . 17
2.2.4.3 Improved models performances . . . 17
2.2.4.4 Results and discussion . . . 18

2.3 DAN and LSTM-based event model . . . 21
2.3.1 Long- and Short-Term Memory (LSTM) networks . . . 21
2.3.2 DAN and LSTM-based architecture . . . 21
2.3.3 Model performance . . . 22

**3 Anomaly detection** **24**
3.1 Detecting anomalies using the loss . . . 24
3.2 Evaluating anomaly detection using labled data . . . 24
3.2.1 Receiver Operator Characteristic (ROC) curve . . . 25
3.2.2 Area Under the Receiver Operator Characteristic Curve (AUC ROC) . . . 25
3.2.3 Use of AUC ROC on our data and results . . . 26

**Conclusion and discussion** **28**
1 Conclusion on the realized work . . . 28
2 Personal conclusions on the project . . . 29
2.1 Esteban Marquer . . . 29
2.2 Prerak Srivastava . . . 29

**Appendices** **31**

**A BULL-ATOS dataset** **33**
A.1 Source of the data . . . 33
A.2 Structure of the log lines . . . 33
A.3 Pre-processing . . . 34
A.4 Usage . . . 34

**B Log lines distribution in the LANL corpus** **35**

**C Detailed architecture of the DAN-based model** **36**
C.1 Initial single-line implementation . . . 36
C.2 Single-line improved implementation . . . 38
C.3 Multi-line implementation with attention . . . 40

**D Detailed architecture of the DAN and LSTM-based model** **42**

# Introduction

Logs are an important part of any computer ecosystem today but, to understand and make future decision on these logs is an hard and important task. Deep learning has emerged as a key player to solve this problem due to its state-of-the-art performance on major tasks related to anomaly detection. Among those tasks, there is intrusion detection, denial of service (DoS) attack detection, hardware and software system failures detection, and malware detection.

Our project centers on anomaly detection in large amounts of system logs, with very few occurrences of said anomalies.

This project was preceded by a one month preliminary internship within the PAPUD project [1]. Part of the data and code comes from the internship, including the BULL-ATOS dataset (see section 1.1, page 2) provided by industrial partners of the PAPUD project.

Usually, when trying to detect anomalies, we train a classifier that detects if a log line correspond to an anomaly or not. However, we can not efficiently train models to detect anomalies due to the lack of said anomalies in the datasets. A viable alternative to a classifier is a predictive model trained to model the "normal behaviour" of the log lines. Using this model, we can detect anomalies as lines differing too much from the modeled "normal behaviour".

The goals of the project can be summarized in three points, which correspond to the three chapters of this report:

- to explore multiple datasets of log lines (see chapter 1, page 2);
- to build predictive deep learning models to model the normal behaviour of log lines and test them on the different datasets (see chapter 2, page 13);
- to explore methods proposed in the literature to detect anomalies using predictive deep learning models (see chapter 3, page 24).

To realize this project, we used exclusively Python (version 3.7)[2], and more specifically the deep learning-oriented library PyTorch[3]. We also used the Pandas[4] and Numpy[5] library to manage raw data.

To train the deep leaning models we required huge computational resources, so we used the computational cluster Grid5000[6] during the project.

# Chapter 1

# Datasets and preprocessing

## 1.1 BULL-ATOS dataset

In the initial stage of the project we used data provided to us by industrial partners of the project (BULL-ATOS [7]), which was used in a preliminary project as mentioned in introduction (page 1). As we did not use it much during this project, we will not say much about this dataset here, except for the pre-processing step, as it had an impact on the pre-processing of the LANL dataset (see subsection 1.2.3, page 3). It is however described in a more extensive way in appendix A (page 33).

### 1.1.1 Pre-processing

We try to apply as little pre-processing on the data as possible. The pre-processing was designed in the preliminary project and works as a pipeline, pre-processing examples on-the-fly using multiple computing threads. This was designed with the usage of large amounts of raw data in mind, without storage of the pre-processed data.

This on-the-fly pre-processing worked with a low enough cost for the amount of data used in the early stages of this project (around 10 million log lines).

### 1.1.2 Usage

This dataset was used to train the first implementation of the DAN-based model (see subsection 2.2.1, page 15).

## 1.2 Los Alamos National Laboratory (LANL) dataset

### 1.2.1 Source of the data

The Los Alamos National Laboratory (LANL) cyber-security dataset (publicly available, see [8]) contains around one billion log lines which is generated over the span of 58 consecutive days. The logs are about anonymized authentication information from Microsoft Windows-based computers and servers [9].

Attacks are executed by a "Red Team" to produce well-defined compromise events. We refer to the log lines corresponding to those attacks as *Red Team Events* or *Red Lines*.

A distribution of those Red Team Events is described in Figure 1.1 (page 3).

Additional information about the distribution of the log lines distribution can be found in the figures of appendix B (page 35).

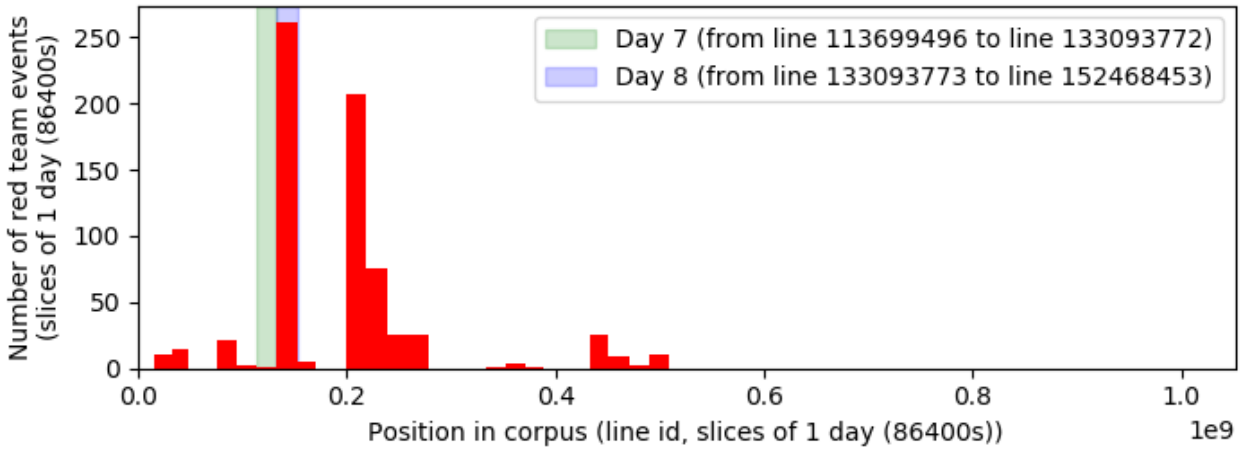



Figure 1.1: Distribution of Red Team Events over the billion of log lines of the dataset

| | |
|---|---|
| Timestamp | `1` |
| Source user | `C6@D1` |
| Destination user | `U7@D2` |
| Source PC | `C6` |
| Destination PC | `C6` |
| Authentication type | `Negotiate` |
| Logon type | `Batch` |
| Autentication orientation | `LogOn` |
| Success/Failure | `Success` |
| `1,C6@D1,U7@D2,C6,C6,Negotiate,Batch,LogOn,Success` | |

Table 1.1: Log line example from the LANL dataset [9]

### 1.2.2 Structure of the log lines

According to [8], the log lines are composed of the following elements: "*time*, *source user@domain*, *destination user@domain*, *source computer*, *destination computer*, *authentication type*, *logon type*, *authentication orientation*, *success/failure*".

An example from the article [9] is presented in Table 1.1.

### 1.2.3 Pre-processing

The data is split into two files, one containing the log lines themselves and another containing information allowing to identify the Red Team Events in the log lines file.

We do very little pre-processing on the data from the LANL database, with two very simple processes. First, we normalize all the lines to a length of 128 characters, by removing the excess characters and padding the shorter lines with a specific padding character. Then, we map of all the characters to numbers using a pre-built character to number mapping, that we call the *dictionary*. That dictionary maps unknown and rare characters to an Out-Of-Vocabulary character, and defines the padding character as an extra character.

Initially, the pre-processing was done on-the-fly, as described in subsection 1.1.1 (page 2). However, even if very flexible, this method is dramatically slowing down the training process of the model when the scale of the data increases. Thus, the datasets described in subsection 1.2.4 were completely pre-processed beforehand. The resulting data was stored in gzip-compressed CSV files.

Together with those CSV files, TXT files containing the relative position of red lines events in each dataset were produced to allow easy manipulation of the pre-processed data.

### 1.2.4 Usage

Models were trained on this dataset in an unsupervised manner, meaning the Red Team Events labels were used only to evaluate the anomaly detection ability of the models (see chapter 3, page 24).

We use this dataset to test our DAN model (see section 2.2, page 15) on intermediate- to large-sized datasets with a simpler structure than the BULL-ATOS dataset we used previously. The two original objectives leading to the use of this dataset were to use a simpler dataset to improve our model, and to compare the performance of our model on the LANL and BULL-ATOS datasets.

Two subsets of the whole dataset were used during the project.

#### 1.2.4.1 *Percentile* corpus

The first one, designated as *percentile*, is composed of three subsets, built with randomly selected slices of the last 40% of the data. This part of the data was chosen because it doesn't contain anomalies. The subsets are as follows:

- a training set, from line 970,000,000 to line 980,000,000 (about 1% of the whole dataset);
- a test set, from line 878,940,000 to line 878,950,000 (about 0.01% of the whole dataset);
- a validation set, from line 713,070,000 to line 713,080,000 (about 0.01% of the whole dataset).

#### 1.2.4.2 *Day 7* and *day 8* corpus

The second subsets of the whole dataset mimic, as strictly as possible with the available information, the use of the LANL corpus in [9], to allow us to easily compare our models with the ones presented in the article.

The data is spread across two days of recorded log lines (see Figure 1.1, page 3).

*Day 8*, the day with the most Red Team Events of the whole corpus (with 261 such events), is used exclusively to analyze the anomaly detection ability of the models (see chapter 3, page 24). It contains 19,374,680 log lines in total.

*Day 7*, the day just before day 8, contains only 1 Red Team Event, and is used for training and evaluating the models. It is split into 3 sets: a validation set containing of the last 2,000 lines, a test set with the 2,000 lines right before those, and the last 19,390,276 lines composing the training set.

Note that in [9] A. Brown *et al* begin the count of days at 0, so *day 7* is the $8^{\text{th}}$ day of the record and *day 8* the $9^{\text{th}}$.

| | |
|---|---|
| Date | 2017-09-08 |
| Timestamp (up to ms) | 02:28:00,017 |
| Server name | CC74.Administration |
| Session ID | no.session.id |
| Workflow ID | no.workflow.id |
| Port address | 0x7A26D608 |
| Tag (undefined) | UNDEF_____ |
| *Envoi de mail*-Begin or -End | Envoi de mail - End |
| 2017-09-08 02:30:00,023 ERROR [stderr] (eService scheduler_ Worker-5) [SEVERE] [CC74.Administration] [*** no.session.id ***] [*** no.workflow.id ***] [0x3F584622] [UNDEF_____] [Envoi de mail - End] | |

Table 1.2: Example of a log line from the BAREM dataset, among those which aren't used to train the model

## 1.3 BAREM dataset

### 1.3.1 Source of the data

We do not have much information about the provider of this dataset, except for the name of the company (BAREM) and the confidential nature of the data. By looking at how the logs are structured we can deduce that they are from an application using a client-server architecture. The logs contain a lot of lines which are most likely not useful to train our model. The data is in general quite varied, and have no pattern that is easily noticeable.

### 1.3.2 Structure of the log lines

First, the log lines showing no connection between a client and the server do not seem to contain any useful information, so we do not use them to train our models. The lines are structured in the 8 following fields: "*date, time stamp (in ms), server name, session id, work flow id, port address, tag,* Envoi de mail-*end or* Envoi de mail-*begin*".

The data we used to train our system follows a similar structure, with some additional fields: "*date, time stamp (in ms), server name, session id, work flow id, port address, tag, name of the task executed by the application, the stack of all the tasks,* `Entry_Point` *or* `Exit_Point`". You can see an example of this structure in Table 1.3 (page 6).

| | |
|---|---|
| Date | 2017-09-08 |
| Timestamp (up to ms) | 02:28:00,017 |
| Server name | CC74.Administration |
| Session ID | no.session.id |
| Workflow ID | no.workflow.id |
| Port address | 0x7A26D608 |
| Tag | START_ACT__ |
| Name of the task executed | Home activity |
| Stack of all the tasks (separated by an exclamation mark) | CitoyenBRWorkflowInstance!<br>homeWorkflowInstance!<br>hp_activity_instance_id |
| Entry or exit point (*entree* or *sortie*) | entree |

```
2018-06-06 07:53:41,090 ERROR [stderr] (default task-5)
 [INFO] [CC74.TS.TransportsScolaires]
 [wGjQrps5TRpcOnQhdXTnBlzE] [14ae7ddb06697d9578dadb77f8b76842]
 [0x4999A60D] [START_ACT___] [Home activity]
 [CitoyenBRWorkflowInstance!homeWorkflowInstance!
 [hp_activity_instance_id] [entree]
```

Table 1.3: Example of a log lines from the BAREM dataset, among those which are used to train the model

| | | |
|---|---|---|
| Session_1 | Workflow_1 | Task_1 |
| Session_1 | Workflow_1 | Task_2 |
| Session_1 | Workflow_1 | Task_3 |
| Session_1 | Workflow_1 | Task_4 |
| Session_1 | Terminated Workflow_1 | |
| Session_1 | Workflow_2 | Task_1 |
| Session_1 | Workflow_2 | Task_2 |
| Session_1 | Ended Session_1 (Using session time out tag) | |

Figure 1.2: Example of a workflow Workflow_2 that is not terminated even if the session is terminated

| | | |
|---|---|---|
| Session_2 | Workflow_3 | Task_1 |
| Session_2 | Workflow_3 | Task_2 |
| Session_2 | Workflow_3 | Task_3 |
| Session_2 | Workflow_3 | Task_4 |
| Session_2 | Terminated Workflow_3 | |
| Session_2 | Workflow_2 (Comes from Session_1) | Task_3 |
| Session_2 | Workflow_2 | Task_4 |
| Session_2 | Ended Session_2 (Using session time out tag) | |

Figure 1.3: Example of how Workflow_2 from Figure 1.2 can restart in Session_2 without any reference to Session_1

#### 1.3.2.1 General structure of the log lines

As represebted in Figure 1.2, Workflow_2 which is not ended but the session somehow ended. So, this Workflow_2 can appear in other sessions and can continue there but without any reference from which session it usually come from (see Figure 1.3).

#### 1.3.2.2 Analysis of the dataset and descriptive statistics

An in-depth analysis of the dataset allowed us to rise the following points of interest about the structure of the data.

- Every new session implies a new Session_ID but also a new Workflow_ID.
- Some sessions disappear without properly ending the ongoing session not the ongoing Workflow_ID.
- In any ongoing session, a Workflow_ID which was not ended properly in previous sessions can appear in the ongoing session without any reference to the session it started in.
- Some sessions do not have any Workflow_ID, and some do not contain logs (or actions); those are empty sessions.

We have done some statistical analysis on the BAREM dataset, leading to the following plots.

- The Figure 1.4, Figure 1.5 and (page 9) present the relation between the total time per session (which is addition of all the delta timestamps per log line, see subsubsection 1.3.3.1, page 12) on the X-axis and the number of log lines per session on the Y-axis.
  - The Figure 1.4 shows the session which doesn't finish in a correct way or which doesn't have a `SESS_TIMEOUT` tag.
  - The Figure 1.5 is dedicated to sessions that have ended correctly with an end session tag.
- The Figure 1.6 (page 9) shows outliers among the sessions which have ended correctly with end session tag (see Figure 1.5). The X-axis shows the delta time stamp when the `SESS_TIMEOUT` tag occur in the outlier and the Y-axis the number of log lines per session.
- The Figure 1.7 (page 10) give the distribution of all the tags in the dataset. It is interesting to note that there are total of around 200 session which have errors out of the 15,000 sessions of the dataset.

  The tags like `BUSINESS_ERR`, `SYNTAX_ERR__`, `EXCP_STA___`, `EXCEPTION__` are considered exception tags.
- The Figure 1.8 (page 10) depicts the distribution of the number of log lines per session, with the outliers on subplot (b) and the other sessions on subplot a().

This analysis did not allow us to find a particular pattern which can help us determine if a session that a user have used to access the application can be called a successful session or not. Thus, we cannot produce anomaly annotations on the log lines or the sessions without getting further information from the source company.

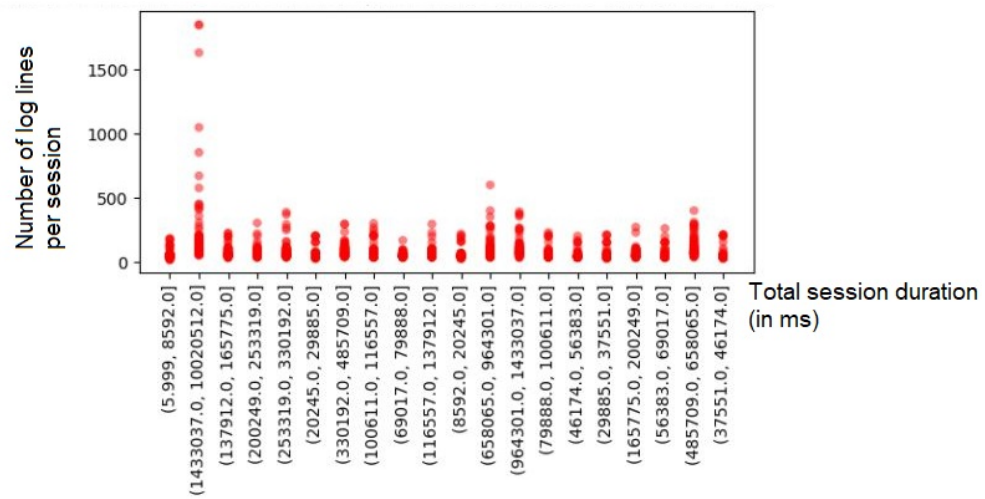



Figure 1.4: Session which have not ended correctly

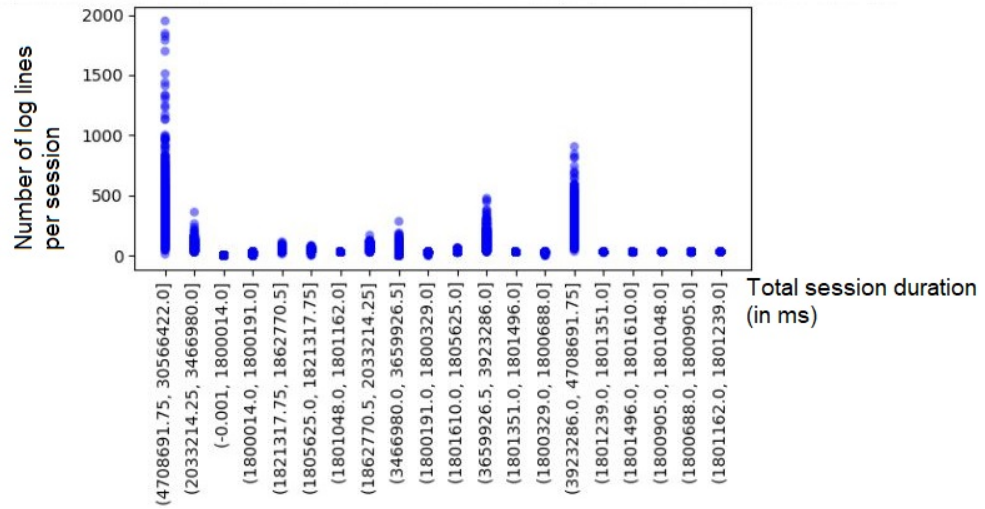



Figure 1.5: Session which have ended correctly

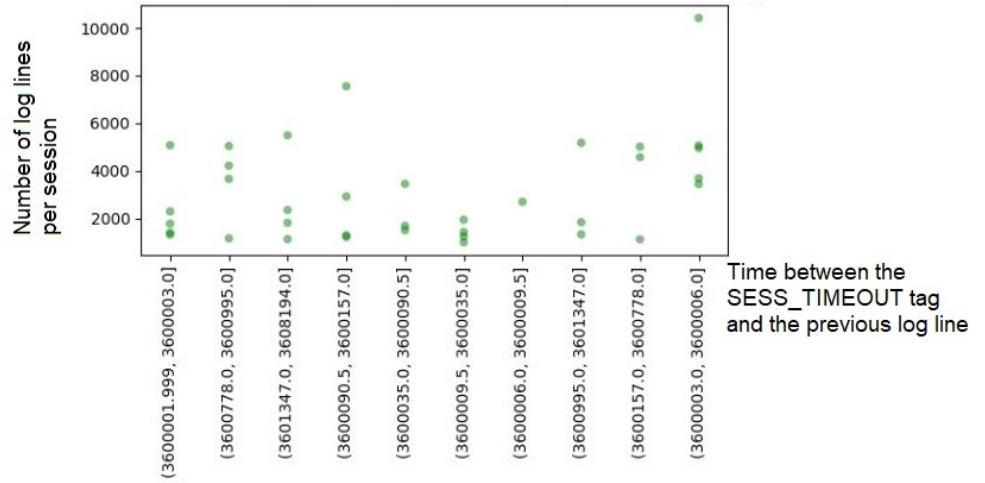



Figure 1.6: Session which have ended correctly but are outliers

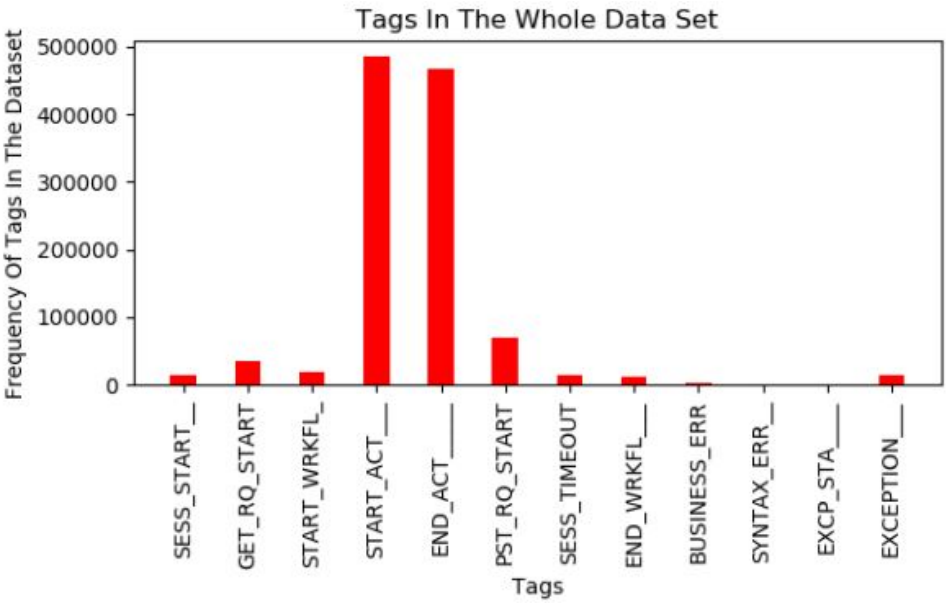
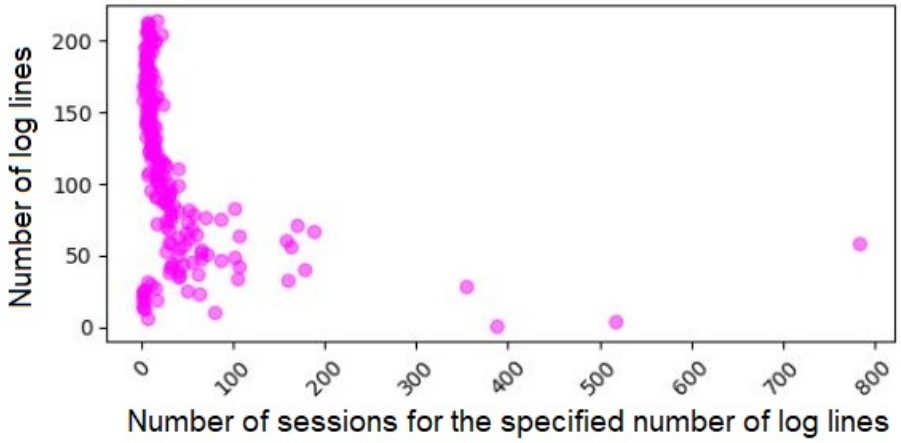
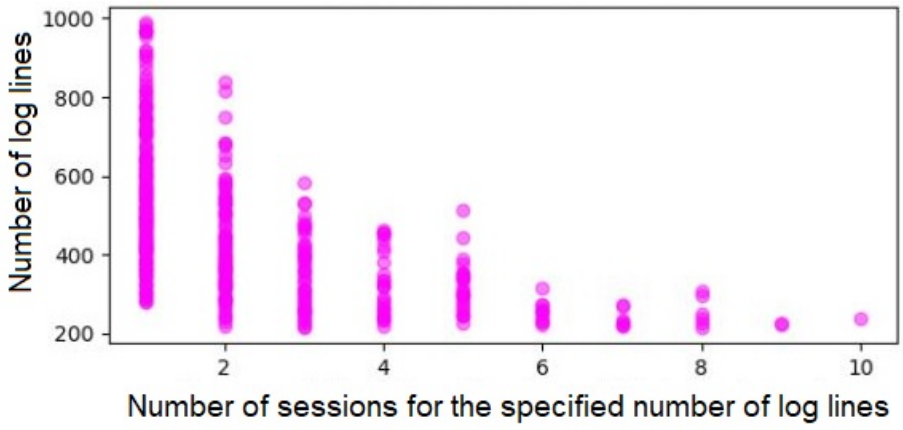



Figure 1.7: Tag distribution in the whole dataset




(a) Main sessions




(b) Outliers

Figure 1.8: Distribution of the number of logs lines per session

### 1.3.3 Pre-processing

Heavy pre-processing needs to be done in this database to extract only the relevant information from the log lines we are interested to work with. The following points describe the sequence of operation composing the pre-processing of the BAREM dataset.

- First, we remove the log lines which contain these strings `***no.session.id***` and `com.eservice.main`. These strings denote when the server is not occupied or when there is no client-server communication. After removing these log lines the log lines containing relevant data are left. At the end of this step, we store the filtered data in a big TXT file.

- We then group the log lines by `Session_ID`. Of all the fields of the log lines, we have only used the following: their `Time Stamp`, `No.session.id`, `tag` and `name of the task executed`.

  We store the log lines in the data structure described in Figure 1.9 (page 12). In the structure, the data field `Data_#` stand for the `name of the task executed`. We join the different fields of each log line into a single string using `^` as a separator.

- We only keep log lines with more than 15 characters, because shorter lines do not contain any relevant data according to our analysis.

- Then we process log lines containing the tags `GET_RQ_START`, `PST_RQ_START`, `SESS_TIMEOUT` or `EXCP_STA____`. Because the data in these tags are either IP address or `null`, we replace said data with the string `------` and include the corresponding lines in our final dataset.

- We also process log lines containing the tags `EXCEPTION___`, `END_WRKFL___` and `END_ACT_____`. We replace the data in those log lines by:

  1. `------` for the `EXCEPTION___` tag, which correspond to exceptions;
  2. `Terminated Workflow Id` for `END_WRKFL___` tag, instead of the workflow identifier originaly contained in the data;
  3. `END_ACT` for `END_ACT_____` tag.

- Note that we keep the timestamp for every log lines because we will use it to calculate the delta time stamp (see subsubsection 1.3.3.1, page 12).

- After doing the above steps we have the whole dataset. We create a *dictionary* as in subsection 1.2.3 (page 3). We use a specific processing for some parts of the log lines:

  1. for every tag in a log line, we consider the whole tag as one entity in the dictionary rather than splitting it into characters;
  2. the data and the timestamps are split into characters and processed by the dictionary.

- We normalize the log lines to a length of 165, because in-depth study of the dataset showed that except for three outliers (two with 243 and one with 320), all the log lines are less than 165 characters. If a log line is shorter, we add a padding character to make it 165 characters long.

- Note that we do not use the time stamps when we create the tensor (see footnote 2, page 14) for every log line. The timestamps are integrated later in the computation (see subsubsection 1.3.3.1, page 12).

```
{
  "Session_Id 1" :  [ "Δts ^ Tag_1 ^ Data_1", "Δts ^ Tag_2 ^ Data_2", ...],
  "Session_Id 2" :  [ "Δts ^ Tag_1 ^ Data_1", "Δts ^ Tag_2 ^ Data_2", ...]
}
```

Figure 1.9: Structure of stored data (Δts stands for delta timestamp)

#### 1.3.3.1 Calculating delta timestamps

We calculate delta time stamps (the difference between timestamps) for every log line with respect to the previous log line. Delta time stamps are strings of characters, so we use Numpy library `np.datetime64(str)` to convert the timestamp string into manipulable time objects, which then can be used to calculate the time difference between two logs. The timestamps are in milliseconds.

Then we just calculate the difference in millisecond from its previous log lines and append it to the tensor of that particular log line before feeding the data to the LSTM layer.

Finally, we normalize the timestamps using an simple normalization method called as *min-max feature scaling technique*.

### 1.3.4 Usage

We can consider training on this dataset as unsupervised (because we have completely unlabeled data), but the heavy pre-processing requiring human expertise makes it more semi-supervised.

Hence, our earlier task which was to predict if the user is satisfied with the usage of the application or not satisfied is still on hold because we cannot find any particular pattern which can distinguish if in a session that a user have used to access the application can be called a successful session usage or not.

The data obtained from the pre-possessing is used to train the model based on the combination of DAN and LSTM, defined section 2.3 (page 21). As described in the subsection 1.3.3 (subsection 1.3.3) we group all the log lines using their session identifier.

Hence, each session is a group of log lines sharing a session identifier. The whole dataset contains `14366` different sessions. We store the pre-processed data in a large file named as `prepro.txt`. We use this file to feed the model with batches, each contains multiple sessions.

Those batches are converted into tensors and concatenated to the calculated delta timestamps.

The data is split into the following subsets:

- a training set, containing 70% of the sessions;
- a validation set, containing 20% of the sessions;
- a test set, containing 10% of the sessions.

# Chapter 2

# Predictive deep-learning models

## 2.1 General informations on our models

We used two models for the project. The first one is a direct implementation of the Deep Averaging Network (DAN) presented in [10] (see section 2.2, page 15), while the other adds a LSTM in the architecture (see section 2.3, page 21).

### 2.1.1 Predictive event models

As explained in the introduction (page 1), we can not efficiently train models to detect anomalies due to the lack of said anomalies in the datasets.

The alternative that was chosen was to use predictive models to predict the normal behaviour of the logs, and to detect the anomalies by comparing them to this normal behaviour.

To achieve this, we try to predict the next line of a sequence of log lines. We use two variants of this approach during the project: predicting one log line using either the previous log line or multiple previous log lines.

In [9], they call such models *event models*, as it models the sequence of events represented by log lines.

### 2.1.2 Character embeddings

Until recently, the majority of neural network architectures used for textual Natural Language Processing were typically used with word embeddings, which are real-valued vector representations of words. However, current literature shows increasingly (for example [9]) that models trained using character embeddings (real-valued vector representations of characters) can achieve similar performance than with word embeddings. Also, character embeddings manage out-of-domain data way better (it is way less likely to encounter an unknown character than to encounter a new word). Thus, following the previous work on the PAPUD project (see the introduction, page 1) and by the request of our supervisor, the models developed during the project use character embeddings and character-level predictions.

This choice has repercussions on how the datasets are pre-processed (see chapter 1, page 2): with word embeddings it is necessary to tokenize the log lines into tokens, decide on how to process punctuation, ..., while with character embeddings none of it is necessary.

### 2.1.3 Loss and model training

In deep learning, we build models to achieve a variety of tasks. Each of those models produce an output.

To achieve the expected results, we compare the produced output and the expected output. A *loss* is a measure of the difference between the two. It is, in specific circumstances, called the distance between the expected and produced outputs.

The training is the process of trying to minimize the loss, to obtain a model that produce outputs as close as possible to the expected output.

To be able to minimize the loss, we need to be able to know the impact of each *parameter*[1] of the model in the loss, before updating them in order to reduce the loss.

The algorithm called *gradient back-propagation* allows us to achieve this.

#### 2.1.3.1 Cross entropy loss and model output

The models we develop are trained to predict log lines; as we are using characters-level predictions (see subsection 2.1.2, page 13), our models need to output a prediction on the possible characters (the characters in the *dictionary*, see subsection 1.2.3, page 3) for each predicted character.

In practice, we use *cross entropy loss*, coupled with a specific output shape for our model predictions.

> Cross entropy indicates the distance between what the model believes the output distribution should be, and what the original distribution really is. [...] Cross entropy measure is a widely used alternative of squared error. It is used when node activations can be understood as representing the probability that each hypothesis might be true, i.e. when the output is a probability distribution. [11]

As specified in the previous description, the output of our model must be a probability distribution over the characters in the *dictionary*. There will be one such distribution per predicted character. Thus the output shape of our model: a two dimensional tensor[2] of dimentions $number\_of\_predicted\_characters$ and $number\_of\_characters\_in\_the\_dictionary$.

Usually, we need to normalize the output of the model to obtain a true probability distribution using an operation called *softmax*. It is a "function that takes as input a vector of K real numbers, and normalizes it into a probability distribution consisting of K probabilities" [12].

However, the PyTorch documentation specifies that their implementation of cross entropy loss already integrates of a *softmax* [13], so our models do not explicitly implement a *softmax*.

[1] *Parameters* (also called *weights* in some cases) are values involved in the computation of the output of any machine learning model. The parameters are tuned during what is called the *training* of the model.

[2] A *tensor* is a specific type of vector used in deep learning, which has properties allowing *gradient back-propagation*.

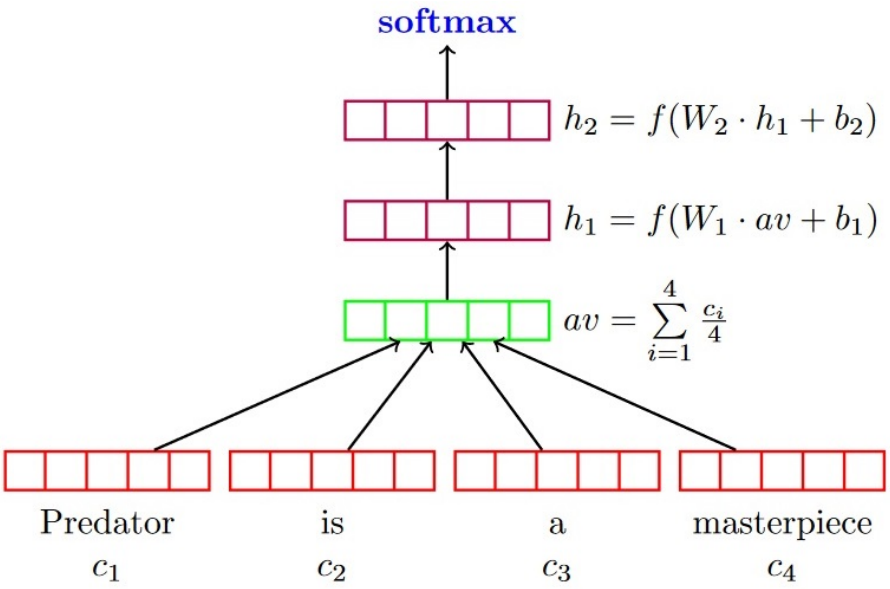



Figure 2.1: DAN architecture applied on a sentence of 4 word embeddings (figure from [10])

## 2.2 Deep Averaging Network (DAN)-based event model

### 2.2.1 Predictive DAN

The Deep Averaging Network is an order-unaware architecture, fast to train and with performance equivalent to state-of-the-art architectures on classification tasks [10, 14]. It is based on the older Neural Bag-Of-Word (NBOW) architecture [15], another order-unaware architecture typically used to represent sentences.

The DAN architecture is simple: it has few layers and contains no recurrence nor convolution. It only contains an averaging function over the input embeddings and a few hidden layers (see Figure 2.1).

The strength of the DAN is its ability to amplify "small but meaningful differences in the word embedding average" [10], allowing it to achieve state-of-the-art performance with, thanks to its simplicity, a computational cost way lower than state-of-the-art architectures (for example the LSTM, see section 2.3, page 21).

It is thus particularly adapted to our usage on the BULL-ATOS and LANL datasets (see section 1.1, page 2 and section 1.2, page 2), as the short training time of the DAN architecture allow us to take advantage the large amount of available data. Comparatively, other state-of-the-art architectures like the LSTM concentrate on achieving high performance with low amounts of data.

NBOW and DAN architectures are typically used with word embeddings (which are real-valued vector representations of word). However, as explained in subsection 2.1.2 (page 13), our DAN model uses character embeddings.

We use the description in [10] as a reference for our implementation of the DAN architecture. However, we use a variant of the Pooling layer, as we use a max pooling instead of an average pooling.

To apply this architecture to our prediction task, the output of our model is a representation of a line.

An interesting feature of the model is its ability to adapt to inputs of different sizes (as it first computes an average on all the input data). Yet, such a property was not exploited in this project, and may be explored in future works.

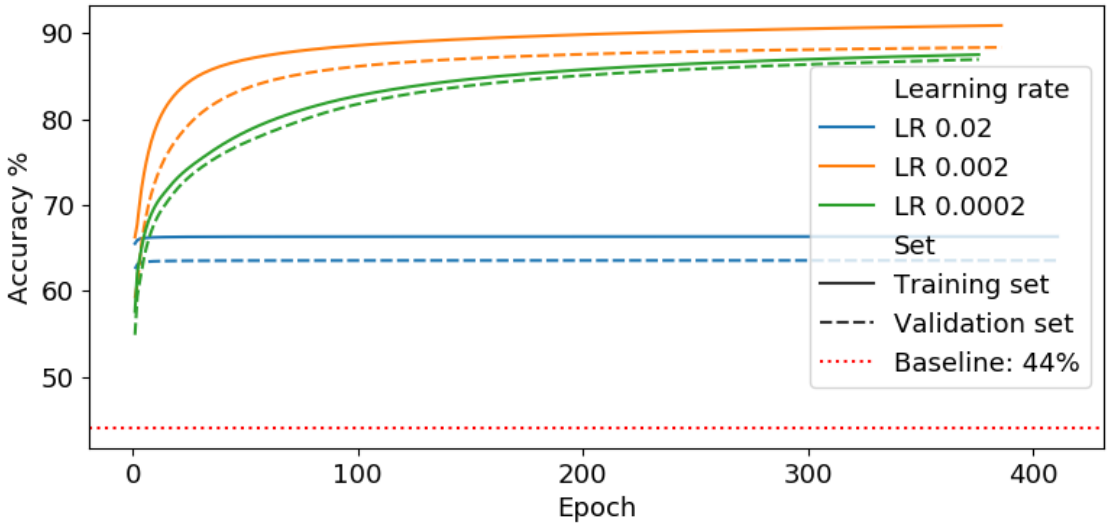



Figure 2.2: Accuracy of the DAN model on the BULL-ATOS dataset, with varying learning rates

### 2.2.2 Performance on the BULL-ATOS dataset

The initial implementation of the DAN model was a single line event model (see subsection 2.1.1, page 13). It was implemented during the internship as a preliminary study and was updated to integrate the necessary hidden layers. The full structure of the model and the number of parameters is presented in appendix section C.1 (page 36).

The accuracy obtained while predicting only the most frequent character (namely the padding character) is used as baseline (around 44% of accuracy) for the experiments on the BULL-ATOS dataset. The result of those experiments is shown in Figure 2.2.

The best accuracy achieved on the BULL-ATOS dataset is nearly 90% on the validation set and 92% on the training set. This result extremely encouraging, especially when compared to our baseline of about half this accuracy: it shows the model manages to learn and is not just randomly outputing characters or outputing only the most frequent character. Also, in a more general (and subjective) perspective, achieving this accuracy with an order-insensitive and barely tuned model is a good result.

### 2.2.3 Adaptation to the LANL dataset

When the DAN model was used as-is on the LANL *percentile* dataset, the performance was comparatively bad: around 67% at best on the training set, and around 62.5% on the validation set. When it was tested on the larger LANL *day 7* dataset, the accuracy was similar, with however around 2% of improvement on the validation set. Also, as shown on Figure 2.3 (page 17), there is no hint of divergence even after more than 50 epochs. This is most likely a typical case of under-fitting, meaning the model is not complex enough to capture the complexity of the data and thus have trouble to learn anything.

### 2.2.4 Attempts to achieve better accuracy and to avoid under-fitting

The most straightforward way to reduce under-fitting is to increase the number of parameters of the model, another way being altering the architecture used. Those are most likely not the only ways of reducing under-fitting, but those two ways were tested duringt the project.

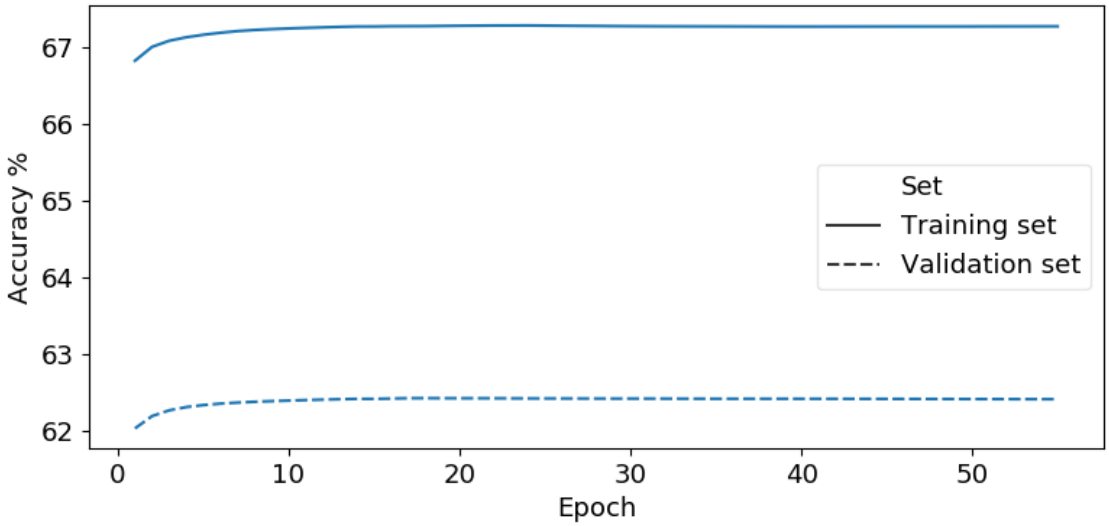



Figure 2.3: First accuracy results of the DAN model on the LANL *percentile* dataset

#### 2.2.4.1 Model with increased parameters

To solve the under-fitting problem, the number of parameters of the model was increased from 1,089,920 to 42,084,416, by changing the size of the two hidden layers from 128 and 128 to 2,048 and 4,096 (see Figure 2.5, page 19). The full structure of this version of the model and the detailed number of parameters is presented in appendix section C.2 (page 38).

#### 2.2.4.2 Multi-line model with attention over lines

An additional modification was tested, but not extensively enough to provide conclusive evidence on the performance. It was inspired by our second model based on a LSTM which used more than 1 log line as input. The architecture was modified to allow any number of input lines to predict a single line.

Also, we added an attention over the lines (see Figure 2.6, page 20). This attention allows to weight each line depending on their content before merging them into a single embedding, and may allow the model to select the most relevant lines among the provided lines. We did not expect much from the attention, but still integrated it to the model because of the simplicity of its implementation, its low computational cost and the supposed lack of negative impact it could have on the performance of the model. At worst it would not deteriorate the performance of the model, and could even improve it noticeably. More experiments are needed to correctly evaluate the impact of this attention on the model.

The full structure of the attention-equipped model and the number of parameters is presented in appendix section C.3 (page 40).

#### 2.2.4.3 Improved models performances

The two variants of the model were trained in parallel on the LANL *day 7* dataset (see subsection 1.2.4, page 4). The single-line variant, which implements the architecture without attention described in Figure 2.5 (page 19), is referred to as the *1-line to 1-line* model. The multi-line variant was tested with 5 log lines as an input, with a few different learning rates[3], due to loss

[3] The learning rate is a parameter of the training of a deep learning model specifying how much the model changes at each training step. The higher it is, the faster the improvement of the model, but at the risk of missing the optimal model state or having a loss explosion (see footnote 4, page 18).

explosion problems[4]. We refer to this model as the *5-lines to 1-line* model.

The result showed on Figure 2.4 are extremely close to the results on Figure 2.3 (page 17). The accuracy hits 67% at best on the training set, and around 64.5% on the validation set for the single-line model. However, the multi-line model, while more complex (and supposedly more powerful), achieves an accuracy lower by around 2% on both sets. It seems that the improvements we did on the model did not manage to improve its performance, nor to solve the under fitting problem. Moreover, they increased the duration of the training (from 45 minutes to more than 3h per epoch), and in the case of the multi-line model, even deteriorated the performance.

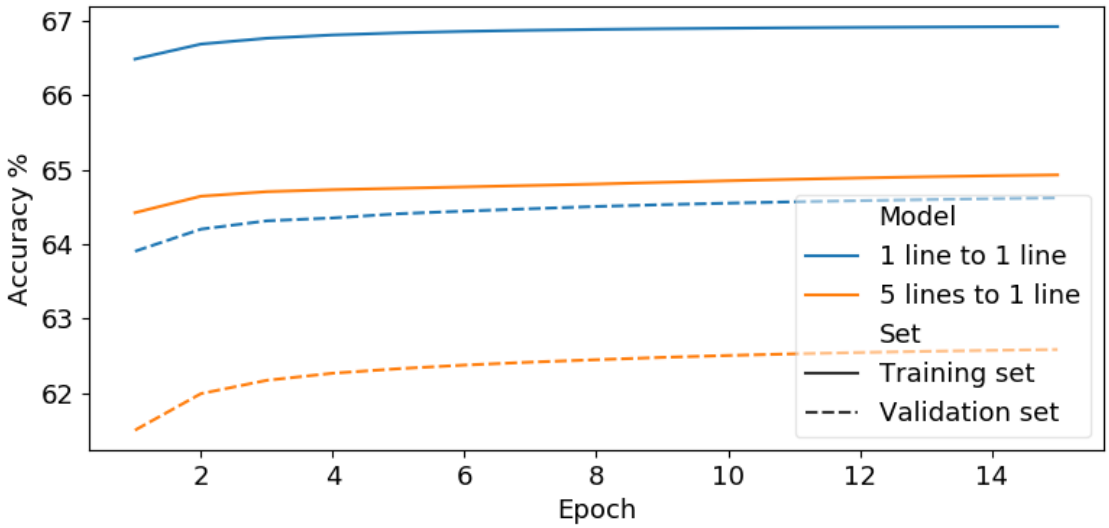



Figure 2.4: Accuracy results of the best *1-line to 1-line* and *5-lines to 1-line* DAN models on the LANL *day 7* dataset

#### 2.2.4.4 Results and discussion

The tests on the multi-line model were not extensive enough to be conclusive, however the results suggest that increasing the number of lines deteriorates the performance. Our hypothesis is that the amount characters on which the max-pooling is done is too much for the model.

When analyzing the results obtained on the LANL dataset, we notice that the accuracy does not improve much from the initialization in the *under-fitting* situations. It can be interpreted as the model not managing to learn.

Also, as shown on Figure 2.2 (page 16), tuning the learning rate of a DAN with few parameters, can bring the accuracy from around 67% to above 90%.

By comparing these two results, it seems more likely that to improve the performance of the DAN model, tuning the learning rate is a lot more effective than the parameter increase we applied.

In conclusion, restoring the model with fewer parameters and extensively tuning the learning rate is more likely to succeed in further improving the performance of the model on the LANL dataset.

[4] The loss explosion is the phenomenon of the loss constantly and rapidly increasing during training, and can happen when the model is too complex or the learning rate is not well tuned.

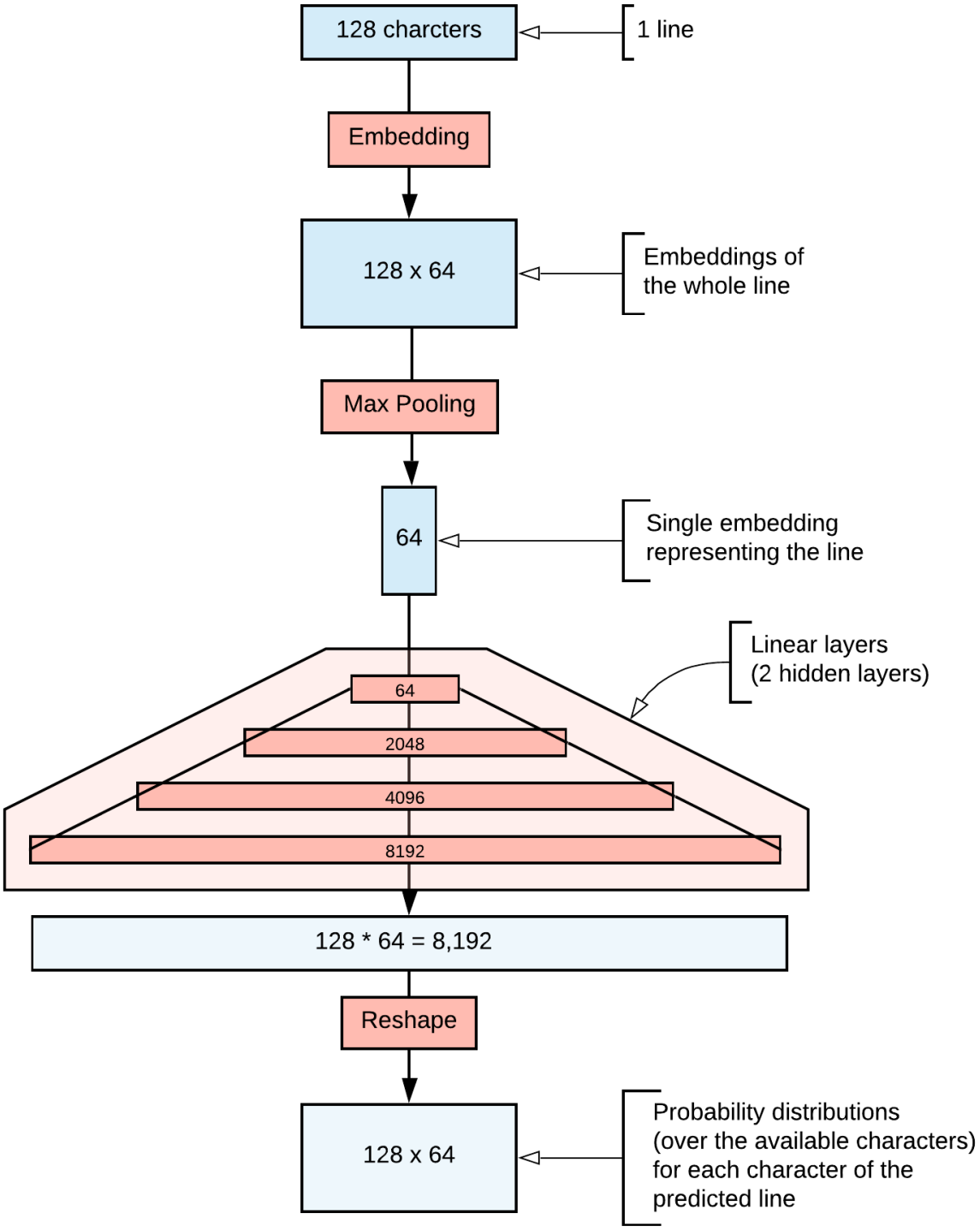



Figure 2.5: Architecture of the DAN model, with increased parameters

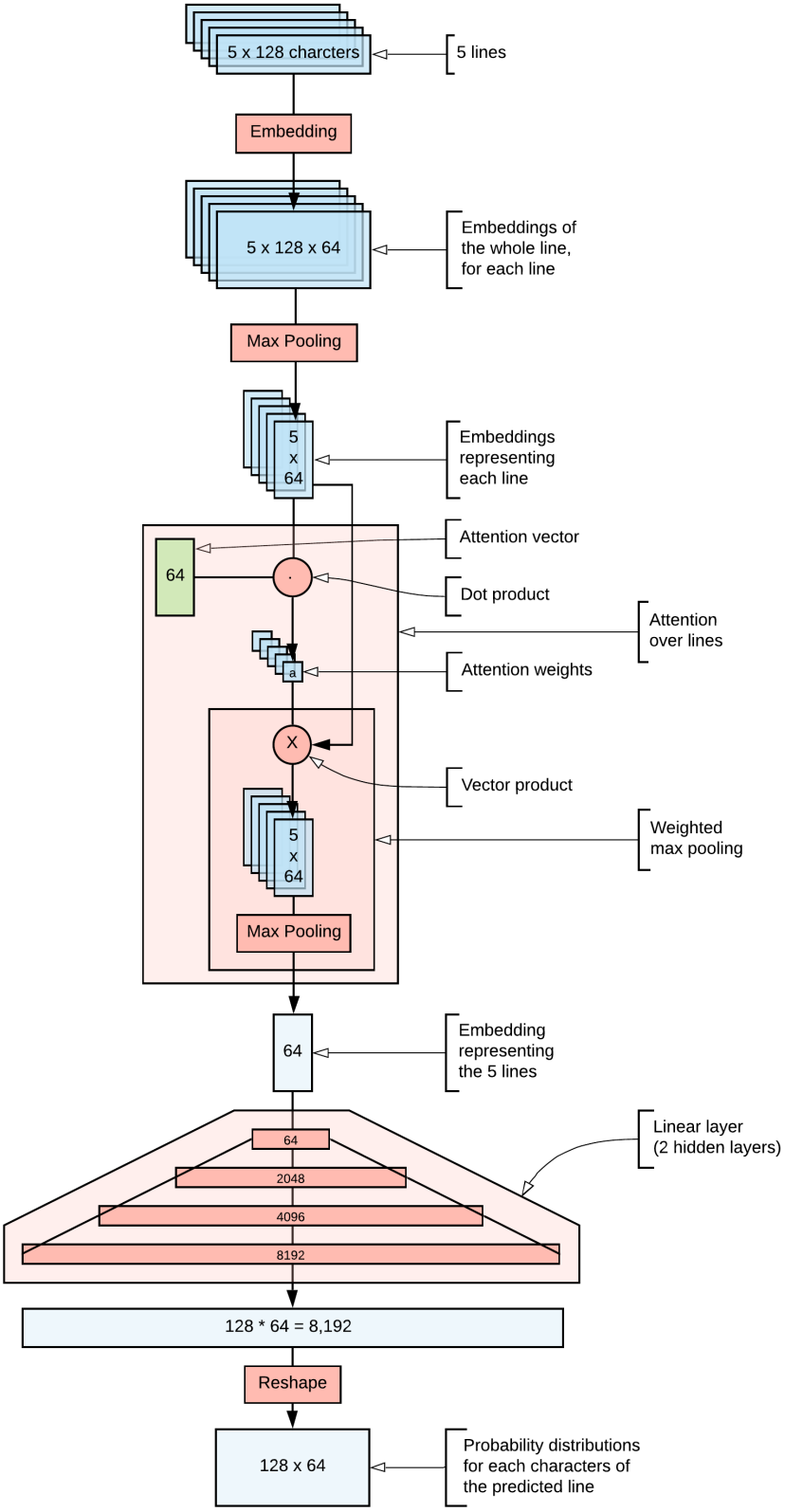



Figure 2.6: Architecture of the *5-lines to single-line* DAN model

## 2.3 DAN and LSTM-based event model

### 2.3.1 Long- and Short-Term Memory (LSTM) networks

A Recurrent Neural Networks (RNN) is a type of neural network adapted to process sequences of inputs. It keeps part of the information from an input to compute the next ones.

Long- and Short-Term Memory (LSTM) networks are an extension for RNN, which basically extends their memory. Therefore it is more powerful than a standard RNN, and can manage longer sequences of inputs (even if this property was not exploited in our setup).

LSTM enable RNN to remember their inputs over a long period of time. This is because LSTM stores their information in an additional memory (called *hidden state*), that LSTM can read, write and delete information from.

This memory can be seen as a gated cell, where gated means that the cell decides whether or not to store or delete information (e.g if it opens the gates or not), based on the importance it assigns to the information it is provided with. The assignment of importance happens through weights, which are also learned by the model.

A LSTM is composed of three gates: input, forget and output gate. These gates determine whether or not to let new input in (input gate), delete the information because it isn't important (forget gate) or to let it impact the output at the current time step (output gate).

### 2.3.2 DAN and LSTM-based architecture

The architecture of the model is shown Figure 2.8 (page 23). It takes two input lines and to predict the next line, and consists of the following components in a sort of pipeline:

- an embedding layer;
- a max pooling layer;
- adding normalized delta time stamps (see subsubsection 1.3.3.1, page 12);
- a LSTM layer;
- a linear layer (somtimes called *perceptron*, or *fully conected layer*).

First, every character from the log line is transformed into embeddings (see subsection 2.1.2, page 13) of 30 features by the embedding layer.

Then, similarly to what was presented for the DAN (see subsection 2.2.1, page 15), a max pooling layer produces one single-dimentional tensor per line (so 2 tensors for the 2 input log lines). Each of those tensors have a size of 30 features, same as the embeddings.

We then concatenate the timestamp of each log line to the corresponding pooling layer output. Those two new tensors represent the pair of log lines.

We now have a sequence of 2 tensors of 31 values.

We pass this sequence to the LSTM, which has a single layer (even if PyTorch allows us to easily stack multiple LSTMs), and has 10 hidden features in its hidden state. The output of the LSTM layer is a sequence of 2 tensors of the size of the hidden feature (10 in our case). We can consider this as a single 2-dimensional tensor of dimensions 2 and 10.

Before feeding the output of the LSTM to the linear layer, we perform a dropout with a probability of 0.2. This means we randomly replace the values by 0; each value has a probability of 0.2 of being replaced. The dropped-out tensor is passed on to the linear layer, that produces a single-dimensional tensor of size $dictionary\_length \times log\_line\_length$ (with $dictionary\_length$ the number of characters in the dictionary, and $log\_line\_length$ the number of characters of our log lines). We reshape this tensor to a 2-dimensional of dimensions

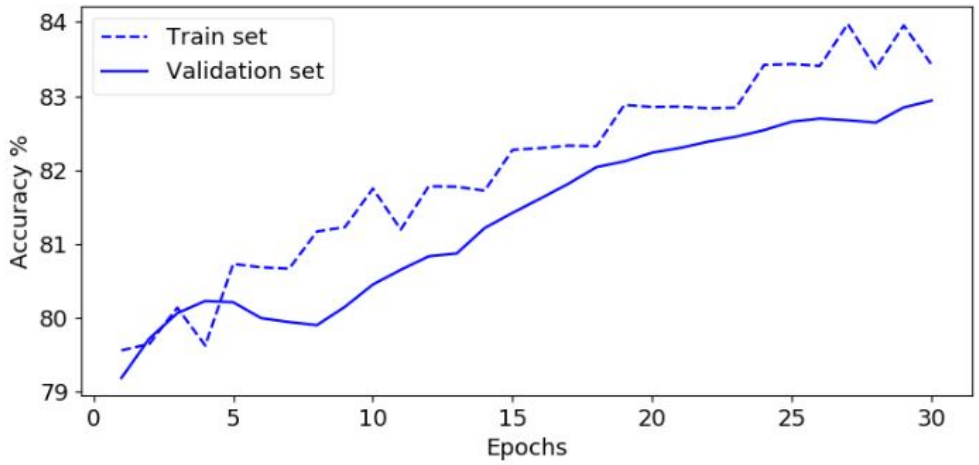



Figure 2.7: Accuracy results of the DAN and LSTM-based model

*dictionary_length* and *log_line_length*, to obtain a distribution over the dictionary for each character as explained in subsubsection 2.1.3.1 (page 14). We can consider that we use the linear layer as a way to produce a usable output from the LSTM output.

It is important that we are taking into account the timestamps, as this feature usually helps the model to yield better results, but we are not predicting the timestamp of the target line. The reason is, predicting the timestamp can be task which is just not useful to our needs and is also irrelevant to our task.

### 2.3.3 Model performance

The performance of the model is calculated on the basis of the predictive accuracy percentage on validation and test sets. This accuracy takes into account the padding character (see subsection 1.3.3, page 11). This may provide an artificial boost to the accuracy.

After doing 30 epochs we have achieved 84% accuracy on train set and around 82.5% accuracy on validation set as shown on Figure 2.7. The data set taken for both validation and test set are randomly picked.

LSTM architecture-based models are usually slow to train, and our model took 3 days to train even with the small size of the dataset and the training algorithm optimizations (for example, multi-GPU training).

The performance of the model seems to be satisfactory in terms of predictive ability as it can predict 84% of the characters correctly. Howaver this usually include the padding characters that we have used to normalize the log lines. It would be interesting to elaborate an accuracy measure that excludes the padding characters, to obtain a "true" measure of the predictive accuracy.

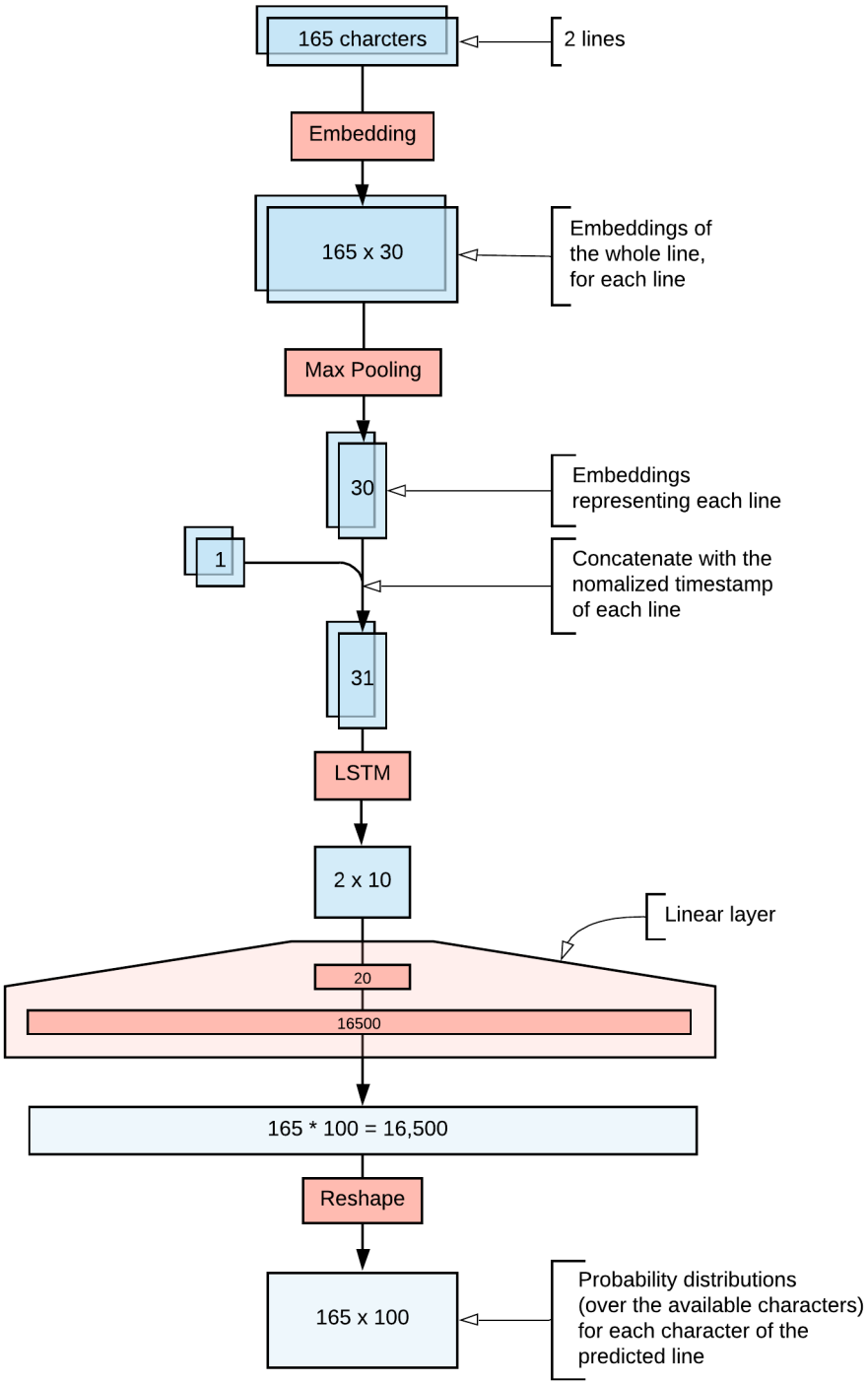



Figure 2.8: Architecture of the 2-lines to single-line DAN and LSTM model

# Chapter 3

# Anomaly detection

## 3.1 Detecting anomalies using the loss

In [9] they propose an out-of-the-box method to detect anomalies using predictive models.

They use the loss (see subsection 2.1.3, page 14) as an anomaly score for each line: they perform a prediction $p_l$ for a line $l$, and compare the prediction $p_l$ and the true line $l$ using the negative log likelihood (NLL) loss $NLL(p_l, l)$. The NLL loss is extremely similar to the cross entropy loss we use, as cross entropy loss is composed of a Softmax and a NLL loss (see subsubsection 2.1.3.1, page 14).

As explained in subsection 2.1.3 (page 14) the loss can typically be interpreted as the distance between the predicted log line and the true log line. The greater the distance, the less likely it is for the true log line $l$ to appear, so the farther the line $l$ is from the standard behaviour of the log line sequence. In other words, "less probable events reciev[e] higher anomaly scores" [9].

To decide which lines are anomalies, it is necessary to put a threshold on the anomaly score (or the loss). If the loss is above that threshold (the distance is greater and the threshold), we consider the line as an anomaly. It wouldn't make much sense to use an arbitrary threshold in those conditions; instead, we test a range of different thresholds (see the next section, section 3.2).

The only difference between the experimental setup of the article [9] and ours is that instead of the NLL loss we use cross entropy loss (see subsubsection 2.1.3.1, page 14).

## 3.2 Evaluating anomaly detection using labled data: Area Under the Receiver Operator Characteristic Curve (AUC ROC)

To evaluate the procedure explained in the previous section (section 3.1, page 24), the authors of [9] use the Area Under the Receiver Operator Characteristic Curve (AUC ROC). This procedure allows to vizualise the performance of the model when using a range of thresholds. More detail about the meaning of the Receiver Operator Characteristic Curve (ROC) are available in subsection 3.2.1 (page 25).

Most of the following explanations on the ROC and the AUC ROC (subsection 3.2.1, page 25) was taken from the clear and concise explanation by Sarang Narkhede [16]. Also, in those explanations, we consider the intuitive meaning of the Area Under a Curve (AUC) as kown.

### 3.2.1 Receiver Operator Characteristic (ROC) curve

The Receiver Operator Characteristic (ROC) curve is one of the many representations of the ability of a binary classifier[1] to distinguish between classes. In our setting, the two classes are *anomaly* and *not anomaly*.

"The ROC curve is plotted with [True Positive Rate (TPR)] against the [False Positive Rate (FPR)] where TPR is on y-axis and FPR is on the x-axis." [16] The TPR is the probability to classify an anomaly as such, while the FPR is the probability to classify a non-anomaly as an anomaly.

We plot those two values (TPR and FPR) for a range of decision thresholds.

Here are a few examples to understand the meaning of those points, written (TPR, FPR):

- at (1, 1) the classifier answers *anomaly* all the time (all the anomalies are correctly predicted, and the non-anomalies are all labeled as anomalies); in other words it accepts everything as an anomaly;
- at (0, 0) the classifier answers *not anomaly* all the time (no anomaly correctly predicted, and no non-anomalies labeled as anomalies); in other words it rejects everything out of the anomalies;
- at (1, 0) the classifier answers *anomaly* for all the anomalies and *not anomaly* for the non-anomalies (it is a "perfectly exact" classifier);
- at (0, 1) the classifier answers *not anomaly* for all the anomalies and *anomaly* for the non-anomalies (it is a "perfectly wrong" classifier); if we invert the classifier output, we obtain a "perfectly exact" classifier.

Note that (1, 1) and (0, 0) are present in the ROC of any classifier (to our knowledge), and represent the extreme values of the threshold.

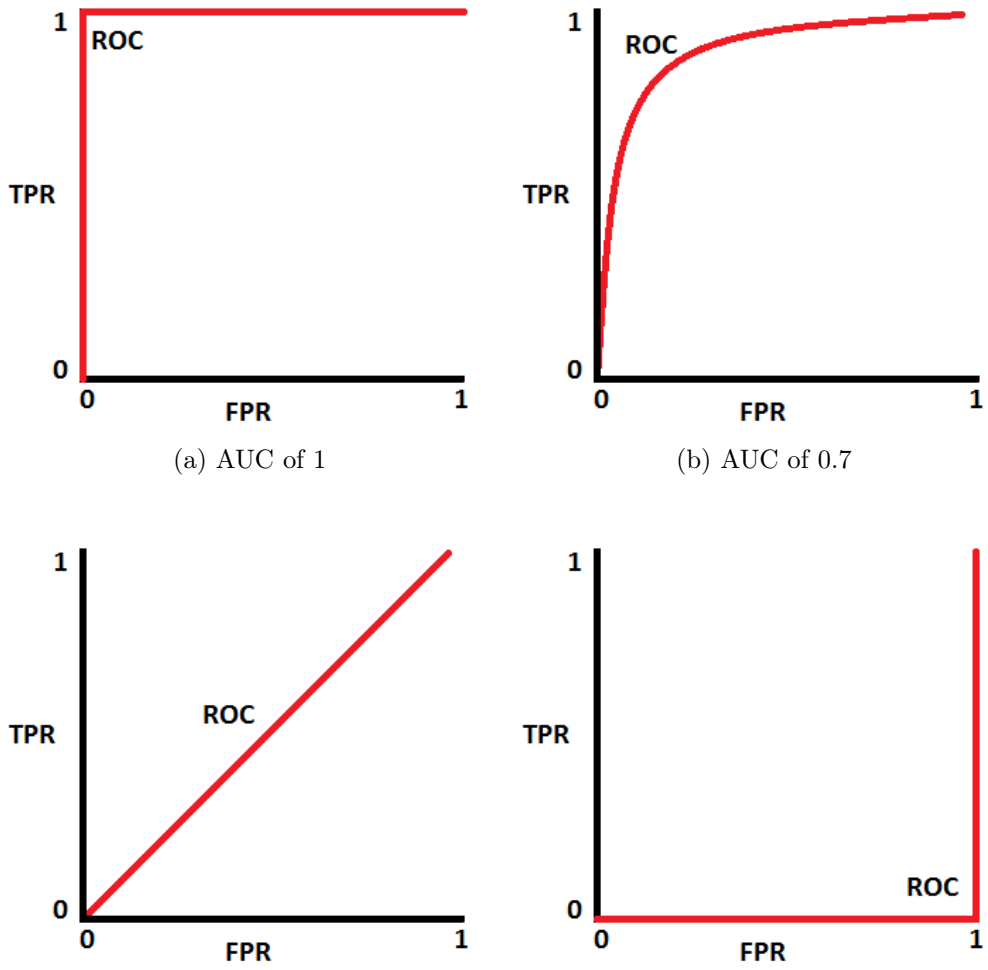
### 3.2.2 Area Under the Receiver Operator Characteristic Curve (AUC ROC)

"[AUC ROC] is a performance measurement for classification problem at various thresholds settings. [...] It tells how much model is capable of distinguishing between classes. Higher the [AUC ROC], better the model is at predicting [if a line is an anomaly or not]." [16] It is a way to interpret the ROC; you can refer to subsection 3.2.1 for an interpretation of individual points.

> An excellent model has AUC near to the 1 which means it has good measure of separability. A poor model has AUC near to the 0 which means it has worst measure of separability. In fact it means it is reciprocating the result. It is predicting 0s as 1s and 1s as 0s. And when AUC is 0.5, it means model has no class separation capacity whatsoever. [16]

Those cases are presented on Figure 3.1 (page 26).

[1] A binary classifier is a classifier that differentiates between two classes. It is equivalent to distinguishing between belonging to a class or not.




(a) AUC of 1

(b) AUC of 0.7

(c) AUC of 0.5

(d) AUC of 0

Figure 3.1: Example of ROC of theoretical models and the corresponding AUC, plots taken from [16]

### 3.2.3 Use of AUC ROC on our data and results

The use of AUC ROC require knowledge of the labels of the log lines (either *anomaly* or not). For the LANL dataset, we use the red line events as anomalies, as explained in section 1.2 (page 2). We use the *day 8* corpus from LANL to compute AUC ROC for the models.

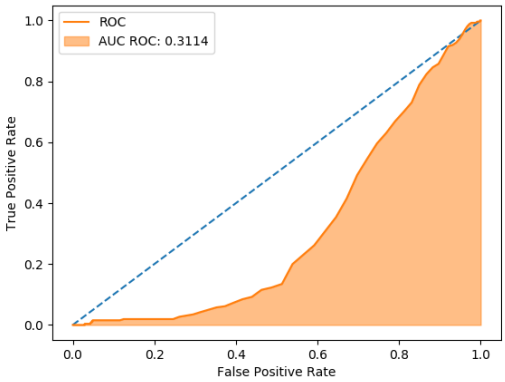
For the two other dataset (BULL-ATOS and BAREM, section 1.1 and section 1.3, page 2 and page 5) we do not have labeled data, it is thus impossible to use AUC ROC on those datasets.

The performance of a first model, trained on LANL *percentile* corpus for 55 epochs, is presented Figure 3.2 (page 27). This model is the one presented in subsection 2.2.3 (page 16), and achieve 67% of predictive accuracy during training and 62.6% of accuracy during validation.

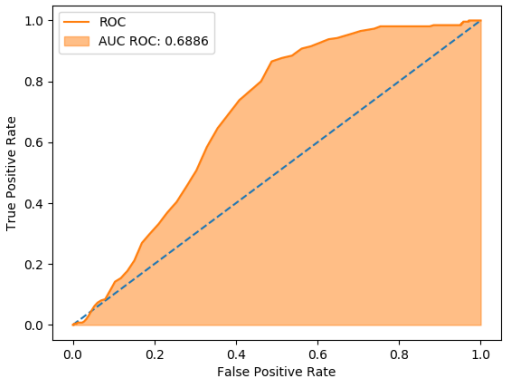
On Figure 3.2 (a), the ROC indicates that the model tend to classify anomalies as non-anomalies, and non-anomalies as anomalies (see subsection 3.2.1, page 25). We obtain a better AUC ROC as on Figure 3.2 (b) if we invert the conclusions of the model. It is equivalent to changing the use of loss higher than the threshold as anomalies to loss lower than the threshold as anomalies.

The performance of the two improved models trained on LANL *day 7* corpus, namely the *1-line to 1-line* and *5-lines to 1-line* models, are presented Figure 3.3 (page 27).

Those models are the ones presented in subsubsection 2.2.4.1 (page 17), and they respec-

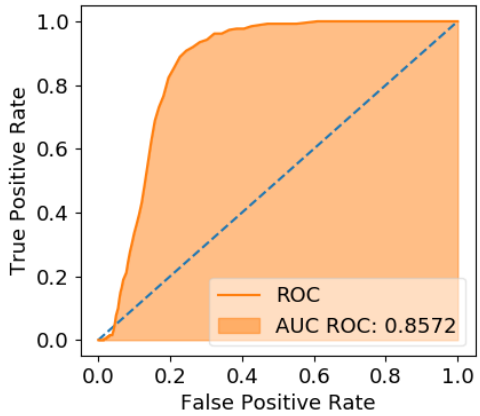



(a) Using loss higher than the threshold as anomalies, as presented in [9] (AUC ROC: 0.31)

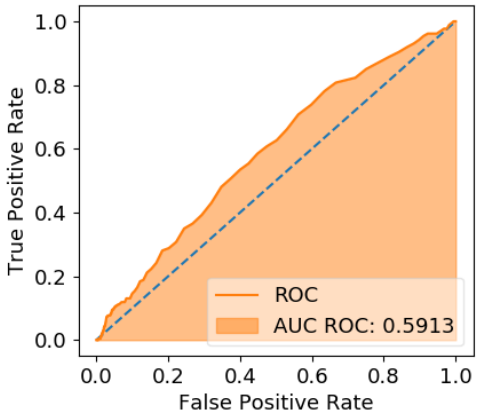



(b) Using loss lower than the threshold as anomalies (AUC ROC: 0.69)

Figure 3.2: AUC ROC of the model trained on LANL *percentile* corpus for 55 epochs (prediction accuracy 67%)




(a) 1-line to 1-line model (prediction accuracy 67%)




(b) 5-lines to 1-line model (prediction accuracy 65%)

Figure 3.3: AUC ROC of the model trained on LANL *day 7* corpus for 15 epochs

tively achieve 67% and 65% of predictive accuracy during training.

The *1-line to 1-line* model achieves an AUC ROC of more than 0.85, (0.15 more than the model trained on LANL *percentile*). This improvement is most likely due to the proximity of the *day 7* to the *day 8* dataset compared to the *percentile* dataset, increasing the resemblance between the normal behaviour on the training and AUC ROC sets. The obtained AUC ROC is still lower than the ones above 95% achieved by the models in [9], but for a much simpler model like the DAN, trained for only 15 epochs moreover, this result is extremely encouraging.

The *5-lines to 1-line* model achieves an AUC ROC of less than 0.6, (0.1 less than the model trained on LANL *percentile*). Also, the ROC is almost the diagonal (see subsection 3.2.2, page 25). Overall, this model is very bad for our detection task.

With a difference in prediction accuracy of only 2%, one is the best and the other is the worst model we produced for anomaly detection.

# Conclusions

## 1 Conclusion on the realized work

To sum up what we achieved, we managed to accomplish all the objectives of the project. We produced predictive models with satisfying performances, explored two datasets other than the initial BULL-ATOS dataset and tested an out-of-the-box method to detect anomalies, achieving good performance with this method. A beneficial part is that by using the same processes and data that in [9], we allow comparability of our results with those of the article. However, we test this method only on one of our datasets due to the lack of labeled anomalies for the other datasets.

The main technical challenge for this project was the duration of the training of the model, which is a standard difficulty in deep learning. However, the amount of data we had to process was larger than usual in textual deep learning, so this point was intensified. Another challenge was the handleing of the data itself, and more specifically the amount of said data, as for the LANL dataset, the storage space required to partially duplicate the data made it a bit troublesome to deal with the pre-processing.

We have at least 5 points we could improve in further steps of the project.

First, the DAN was not tested extensively on the LANL dataset, yet our results on the BULL-ATOS dataset shows that we can greatly improve the results by altering the learning rate (see Figure 2.2, page 16). Further experiments on the LANL dataset could thus bring out the most out of the DAN architecture.

Second, the DAN and LSTM-based model (see section 2.3, page 21) was not tested with the AUC ROC procedure (see chapter 3, page 24), as it was not trained on the LANL dataset. It would be necessary to test this architecture with AUC ROC to be able to compare it with the DAN-based model.

Third, we explored only a single anomaly detection procedure using predictive models. There might be other interesting procedures presented in the literature. Also, we tested only two model architectures, and it would be interesting to widen the range of architectures tested for the PAPUD project.

Forth, the deep learning domain is ever-changing, and the specific area of log analysis is very active. Thus there are always new articles on the subject, which could help improve the results obtained and open new problematics.

Fifth and last point, there is always, in deep learning, a way to improve the performance of the models. In future works, it would interesting to continue tuning the models we have and training them to the limit to obtain better results.

# 2 Personal conclusions on the project

## 2.1 Esteban Marquer

This project was very interesting for me, especially since it is the continuation of the internship I did last summer (the one mentioned in introduction, section 1.1 and subsection 2.2.2, page 1, page 2 and page 16). I appreciated deepening my knowledge about deep learning techniques, for example with the AUC ROC procedure. Also, this project allowed me to improve my Python coding skills.

I think this project matches the first year of Master program quite well, and is also a good introduction to the second year. In my opinion, the most beneficial classes for the project were those about corpus management, data science and machine learning. Those helped me in the predicament I was in when dealing with the pre-processing of the LANL dataset, and more generally helped me improve the quality of my use of the datasets. Moreover, the large amount of classes using Python and the variety of code styles taught help increase my mastery of python (and I noticed that the code I produced during the internship was far from being "pythonic"). Aside from this, English classes allowed me to improve (hopefully) the quality of my writing, making this report a bit clearer than what I could have written without them. To conclude, I am satisfied with what I learned during the project, and would say I am proud to have achieved the results we achieved.

## 2.2 Prerak Srivastava

So, to conclude this report I would like to say that the supervised project on the topic anomaly detection using PyTorch helped in expanding our knowledge in the field of deep learning. Due to this I gained knowledge about architectures like multi-layer perceptron, convolution neural networks, DAN(Deep averaging network), LSTM, linear layers. Apart from this I also learned some deep learning concepts related to the field of NLP which are like character embedding, character level dictionary, max-pooling etc.

Because we have used PyTorch in this project I got a lot of understanding of how an PyTorch works and how to build deep learning pipelines using this framework also I understood a bit of hyper-parameter tuning and how to extract features from the textual data.

This project really helped me in improving my python skills because of large amount of pre-proceesing task that I need to perform for the Barem dataset.

There are few classes in our MSc 1st year program like the data science one which helped me to be good at using Python graph plotting library like matplotlib and to write clean and documented Python code.

Not to forget I learned how to work as team and also with the help of our supervisors I got an understanding on how to approach and bisect a problem and solve it using an deep learning approach.

Hence, this project is a great inclusion in the 1st year of this MSc Program and helped us to expand our knowledge in using the latest technologies and research methodologies in the field of NLP.

# Appendices

# Appendix A

# BULL-ATOS dataset

## A.1 Source of the data

The *BULL-ATOS dataset* is a dataset taken from a preliminary work on the PAPUD project[1], which was part of an internship last summer. The data was provided to us by BULL-ATOS [7], one of the industrial partners of the PAPUD project.

This dataset consists of more than 400 GiB of compressed log files system log files from BULL-ATOS servers (making it a confidential dataset). Due to the large amount of data available, only a small subset of around 10 million lines of this data was used at the time.

The data from that dataset is confidential.

## A.2 Structure of the log lines

The structure of the log lines in the dataset is similar to a lot of standard logging tools (like the `logging` package in Python), with space-separated fields. However, the signification of the fields was not explicitly provided with the data, and the structure presented in Table A.1 (page 33) comes from an analysis of the provided files.

Due to the confidential nature of the data, the example we present in Table A.1 (page 33) has been altered.

| | |
|---|---|
| Timestamp | `1524463200` |
| Date | `2018 Apr 23` |
| Time | `08:00:00` |
| User | `OOO` |
| Process | `authpriv` |
| Message type | `info` |
| Message | `access granted for user root (uid=0)` |
| `1524463200 2018 Apr 23 08:00:00 OOO authpriv info acce[...]ot (uid=0)` | |

Table A.1: Log line example from the BULL-ATOS industrial dataset (the full line has been abbreviated)

## A.3 Pre-processing

We try to do as little pre-processing on the data as possible. The pre-processing was designed in the preliminary project and works as a pipeline, pre-processing examples on-the-fly using multiple computing threads. This was designed with the usage of large amounts of raw data in mind.

The complete pre-processing is split into the following tasks, each of those achieved by a different comuting thread:

1. the Timestamp, Date, Time and User elements are removed, as they are redundant and do not interest us for now;
2. hexadecimal numbers and memory addresses are replaced by a specific character, with a different character for each possible structure of hexadecimal numbers (`0x0005` will map to `<hex4x>`, `ac406005` to `<hex8>`, ...);
3. all the lines are normalized to a length of 200 characters, by removing the excess characters and padding the shorter lines with a specific padding character;
4. the characters are mapped to numbers using an automatically generated dictionary, similar to what is presented in subsection 1.2.3 (page 3).

This on-the-fly pre-processing worked pretty well with a low enough cost for the amount of data used in the early stages of this project (around 10 million log lines).

## A.4 Usage

Multiple variations of the dataset have been created, with mainly alterations on the pre-processing step and the length of the dataset.

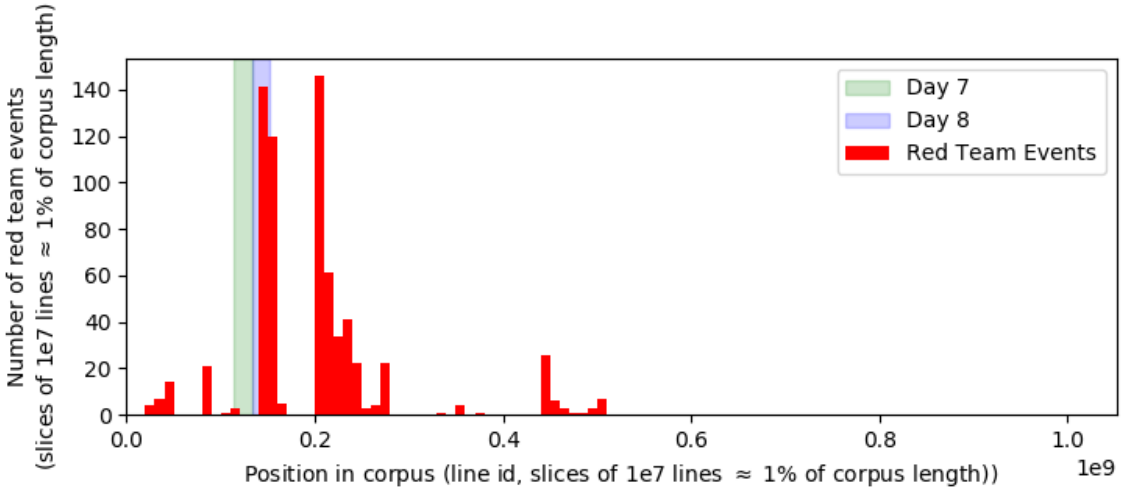
All the dataset variations are generated from the raw data and are composed of three subsets: a test and validation set of 2 000 log lines each, and a training set containing the rest of the data.

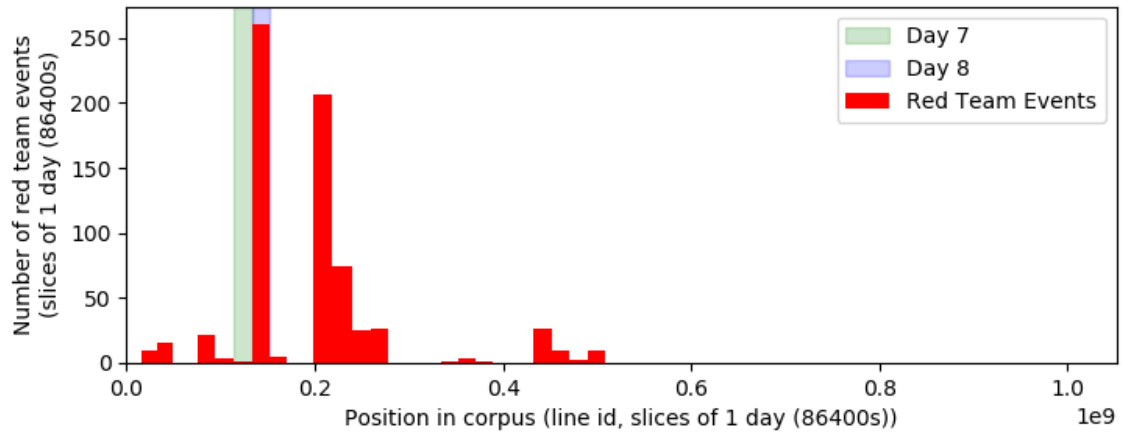
As written in subsection 1.1.2 (page 2), those datasets were used to train only the first implementation of the DAN-based model (see subsection 2.2.1, page 15).

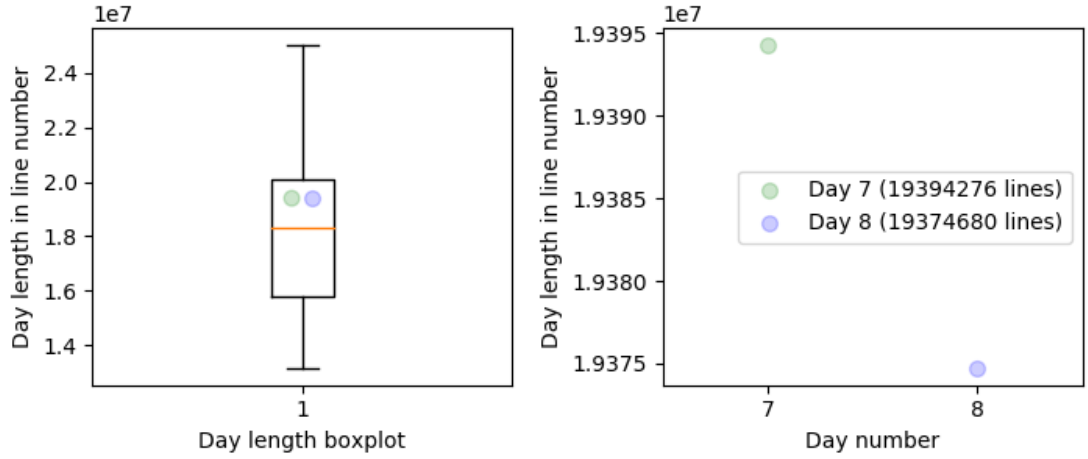
We can consider the training done with this dataset as semi-supervised, as the pre-processing step is tuned after a manual analysis of the data and the performances of the trained model.

# Appendix B

# Log lines distribution in the LANL corpus



# Appendix C

# Detailed architecture of the DAN-based model

This appendix describes the layers and the dimension of the data at each of those layers, for the three variants of the DAN-based architecture presented in section 2.2 (page 15). In the following tables, multiple dimensions are written separated bi commas.

## C.1 Initial single-line implementation

| **Layer** | **Dimension of the data at layer** | |
|---|---|---|
| Input data (128 characters) | $N$ | 128 characters |
| Character dictionary | $N$, dictionary size | 128, 64 |
| Embedding layer (output $E = 128$) | $N$, $E$ | 128, 128 |
| Pooling layer (maxpool) | $E$ | 128 |
| Multilayer Perceptron (layer 1) | $h_1$ | 128 |
| Multilayer Perceptron (layer 2) | $h_2$ | 128 |
| Multilayer Perceptron (layer 3) | $N *$ dictionary size | $128 * 64 = 8,192$ |
| Reshaping | $N$, dictionary size | 128, 64 |
| Softmax | $N$, dictionary size | 128, 64 (probability distribution over the character dictionary) |

Table C.1: Single-line implementation of the DAN model for the BULL-ATOS dataset: size of the data

| **Layer** | **Number of parameters of the layer** | |
|---|---|---|
| *Input data (128 characters)* | | |
| *Character dictionary* | | |
| Embedding layer (output $E = 128$) | input | dictionary size |
| | output | $E$ |
| | weights | $64 * 128$ |
| | **total** | $8,192$ |
| Pooling layer (maxpool) | | |
| Multilayer Perceptron (layer 1) | input | $E$ |
| | output | $h_1$ |
| | bias | 128 |
| | weights | $128 * 128$ |
| | **total** | $16,512$ |
| Multilayer Perceptron (layer 2) | input | $h_1$ |
| | output | $h_2$ |
| | bias | 128 |
| | weights | $128 * 128$ |
| | **total** | $16,512$ |
| Multilayer Perceptron (layer 3) | input | $h_2$ |
| | output | $N *$ dictionary size |
| | bias | 128 |
| | weights | $128 * 128 * 64$ |
| | **total** | $1,048,704$ |
| *Reshaping* | | |
| *Softmax* | | |
| | **total** | $1,089,920$ |

Table C.2: Single-line implementation of the DAN model for the BULL-ATOS dataset: number of parameters

## C.2 Single-line improved implementation

| **Layer** | **Dimension of the data at layer** | |
|---|---|---|
| Input data (128 characters) | $N$ | 128 characters |
| Character dictionary | $N$, dictionary size | 128, 64 |
| Embedding layer (output $E = 64$) | $N$, $E$ | 128, 64 |
| Pooling layer (maxpool) | $E$ | 64 |
| Multilayer Perceptron (layer 1) | $h_1$ | $2,048$ |
| Multilayer Perceptron (layer 2) | $h_2$ | $4,096$ |
| Multilayer Perceptron (layer 3) | $N *$ dictionary size | $128 * 64 = 8,192$ |
| Reshaping | $N$, dictionary size | 128, 64 |
| Softmax | $N$, dictionary size | 128, 64 (probability distribution over the character dictionary) |

Table C.3: Single-line implementation of the DAN model for the LANL dataset: size of the data

| **Layer** | **Number of parameters of the layer** | |
|---|---|---|
| *Input data (128 characters)* | | |
| *Character dictionary* | | |
| Embedding layer (output $E = 64$) | input | dictionary size |
| | output | $E$ |
| | weights | $64 * 64$ |
| | **total** | $4,096$ |
| *Pooling layer (maxpool)* | | |
| Multilayer Perceptron (layer 1) | input | $E$ |
| | output | $h_1$ |
| | bias | 64 |
| | weights | $64 * 2,048$ |
| | **total** | $131,136$ |
| Multilayer Perceptron (layer 2) | input | $h_1$ |
| | output | $h_2$ |
| | bias | $2,048$ |
| | weights | $2,048 * 4,096$ |
| | **total** | $8,390,656$ |
| Multilayer Perceptron (layer 3) | input | $h_2$ |
| | output | $N *$ dictionary size |
| | bias | $4,096$ |
| | weights | $4,096 * 128 * 64$ |
| | **total** | $33,558,528$ |
| *Reshaping* | | |
| *Softmax* | | |
| | **total** | $42,084,416$ |

Table C.4: Single-line implementation of the DAN model for the LANL dataset: number of parameters

## C.3 Multi-line implementation with attention

| **Layer** | **Dimension of the data at layer** | |
|---|---|---|
| Input data (5 lines of 128 characters) | 5, $N$ | 5, 128 characters |
| Character dictionary | 5, $N$, dictionary size | 5, 128, 64 |
| Embedding layer (output $E = 64$) | 5, $N$, $E$ | 5, 128, 64 |
| Pooling layer (maxpool) | 5, $E$ | 5, 64 |
| **Attention over the lines** | | |
| Attention weights (1 weight per line) | 5 | 5 |
| Attention-weighted maxpool | $E$ | 64 |
| Multilayer Perceptron (layer 1) | $h_1$ | $2,048$ |
| Multilayer Perceptron (layer 2) | $h_2$ | $4,096$ |
| Multilayer Perceptron (layer 3) | $N *$ dictionary size | $128 * 64 = 8,192$ |
| Reshaping | $N$, dictionary size | 128, 64 |
| Softmax | $N$, dictionary size | 128, 64 (probability distribution over the character dictionary) |

Table C.5: Multi-line implementation of the DAN model for the LANL dataset: size of the data

| **Layer** | **Number of parameters of the layer** | |
|---|---|---|
| *Input data (128 characters)* | | |
| *Character dictionary* | | |
| Embedding layer (output $E = 64$) | input | dictionary size |
| | output | $E$ |
| | weights | $64 * 64$ |
| | **total** | $4,096$ |
| *Pooling layer (maxpool)* | | |
| **Attention over the lines** | | |
| Attention weights (1 weight per line) | input | $E$ |
| | weights | 64 |
| | **total** | 64 |
| *Attention-weighted maxpool* | | |
| Multilayer Perceptron (layer 1) | input | $E$ |
| | output | $h_1$ |
| | bias | 64 |
| | weights | $64 * 2,048$ |
| | **total** | $131,136$ |
| Multilayer Perceptron (layer 2) | input | $h_1$ |
| | output | $h_2$ |
| | bias | $2,048$ |
| | weights | $2,048 * 4,096$ |
| | **total** | $8,390,656$ |
| Multilayer Perceptron (layer 3) | input | $h_2$ |
| | output | $N *$ dictionary size |
| | bias | $4,096$ |
| | weights | $4,096 * 128 * 64$ |
| | **total** | $33,558,528$ |
| *Reshaping* | | |
| *Softmax* | | |
| | **total** | $42,084,480$ |

Table C.6: Multi-line implementation of the DAN model for the LANL dataset: number of parameters

# Appendix D

# Detailed architecture of the DAN and LSTM-based model

| **Layer** | **Dimension of the data at layer** | |
|---|---|---|
| Input data (2 lines of 165 characters) | 2, $N$ | 2, 165 characters |
| Embedding layer (output $E = 30$) | 2, $N$, $E$ | $2 * 165 * 30$ |
| Pooling layer (maxpool) | 5, $E$ | $2 * 30$ |
| Reshaping | 2, 1, E | $2 * 1 * 30$ |
| LSTM | 2, Batch Size, Hidden Size | $2 * 1 * 10$ |
| Reshaping | | $20 * 1$ |
| Dropout (p=0.2) | | |
| Linear Layer | N, Dictionary Size | $165 * 100$ |

Table D.1: Multi-line implementation of the DAN and LSTM-based model

# Bibliography


[1] *16037 PAPUD*. itea3.org, July 2018. URL: https://itea3.org/project/papud.html (visited on 07/31/2018).

[2] *Welcome to Python.org*. May 2019. URL: https://www.python.org (visited on 04/20/2019).

[3] *PyTorch*. May 2019. URL: https://pytorch.org (visited on 04/20/2019).

[4] *Python Data Analysis Library — pandas: Python Data Analysis Library*. Mar. 2019. URL: https://pandas.pydata.org/index.html (visited on 04/20/2019).

[5] *NumPy — NumPy*. May 2019. URL: https://www.numpy.org (visited on 04/20/2019).

[6] *Grid5000*. May 2019. URL: https://www.grid5000.fr/w/Grid5000:Home (visited on 04/20/2019).

[7] *Produits*. Atos, Aug. 2018. URL: https://atos.net/fr/produits (visited on 08/01/2018).

[8] Alexander D. Kent. *Comprehensive, Multi-Source Cyber-Security Events*. Los Alamos National Laboratory. July 2017. DOI: 10.17021/1179829. URL: https://csr.lanl.gov/data/cyber1.

[9] Andy Brown et al. "Recurrent Neural Network Attention Mechanisms for Interpretable System Log Anomaly Detection". In: *arXiv* (Mar. 2018). eprint: 1803.04967. URL: https://arxiv.org/abs/1803.04967.

[10] Mohit Iyyer et al. "Deep Unordered Composition Rivals Syntactic Methods for Text Classification". In: *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)* 1 (2015), pp. 1681–1691. DOI: 10.3115/v1/P15-1162.

[11] Paras Dahal. "Classification and Loss Evaluation - Softmax and Cross Entropy Loss". In: *DeepNotes* (May 2017). URL: https://deepnotes.io/softmax-crossentropy.

[12] *Softmax function - Wikipedia*. May 2019. URL: https://en.wikipedia.org/wiki/Softmax_function (visited on 04/20/2019).

[13] *torch.nn.CrossEntropyLoss — PyTorch master documentation*. May 2019. URL: https://pytorch.org/docs/stable/nn.html#torch.nn.CrossEntropyLoss (visited on 04/17/2019).

[14] Xilun Chen et al. "Adversarial Deep Averaging Networks for Cross-Lingual Sentiment Classification". In: *arXiv* (June 2016). eprint: 1606.01614. URL: https://arxiv.org/abs/1606.01614.

[15] Jocelyn D'Souza. "An Introduction to Bag-of-Words in NLP". In: *Medium* (June 2018). URL: https://medium.com/greyatom/an-introduction-to-bag-of-words-in-nlp-ac967d43b428.

[16] Sarang Narkhede. "Understanding AUC - ROC Curve". In: *Towards Data Science* (June 2018). URL: https://towardsdatascience.com/understanding-auc-roc-curve-68b2303cc9c5.

[17] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: http://www.deeplearningbook.org (visited on 01/11/2019).

[18] Vladimir Golovko et al. "A Shallow Convolutional Neural Network for Accurate Handwritten Digits Classification". In: *Communications in Computer and Information Science* 673 (Feb. 2017), pp. 77–85. DOI: 10.1007/978-3-319-54220-1_8.

[19] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning Internal Representation by Error Propagation". In: vol. Vol. 1. MIT Press, Jan. 1986.

[20] Ke Zhang et al. "Automated IT system failure prediction: A deep learning approach". In: *undefined* (2016). URL: https://www.semanticscholar.org/paper/Automated-IT-system-failure-prediction%3A-A-deep-Zhang-Xu/95193d2c016f1ee266b1dbf714678ce6bb1bb2

[21] Yoon Kim. "Convolutional Neural Networks for Sentence Classification". In: *arXiv* (Aug. 2014). eprint: 1408.5882. URL: https://arxiv.org/abs/1408.5882.

[22] Jason Brownlee. "A Gentle Introduction to Mini-Batch Gradient Descent and How to Configure Batch Size". In: *Machine Learning Mastery* (July 2017). URL: https://machinelearningmastery.com/gentle-introduction-mini-batch-gradient-descent-configure-batch-size (visited on 04/13/2018).

[23] Christopher Olah. *Understanding LSTM Networks*. Aug. 2015. URL: http://colah.github.io/posts/2015-08-Understanding-LSTMs (visited on 04/24/2018).

[24] Andrej Karpathy. *The Unreasonable Effectiveness of Recurrent Neural Networks*. May 2015. URL: http://karpathy.github.io/2015/05/21/rnn-effectiveness (visited on 04/23/2018).

[25] Pavel Surmenok. "Estimating an Optimal Learning Rate For a Deep Neural Network". In: *Towards Data Science* (Nov. 2017). URL: https://towardsdatascience.com/estimating-optimal-learning-rate-for-a-deep-neural-network-ce32f2556ce0 (visited on 08/17/2018).

[26] Dinghan Shen et al. "Baseline Needs More Love: On Simple Word-Embedding-Based Models and Associated Pooling Mechanisms". In: *CoRR* (May 2018). arXiv: 1805.09843 [cs.CL]. URL: http://arxiv.org/abs/1805.09843 (visited on 08/17/2018).

[27] Rafal Józefowicz et al. "Exploring the Limits of Language Modeling". In: *CoRR* (Feb. 2016). arXiv: 1602.02410 [cs.CL]. URL: http://arxiv.org/abs/1602.02410 (visited on 08/17/2018).

[28] *About SYNALP*. SYNALP, July 2018. URL: http://synalp.loria.fr/pages/about-synalp (visited on 07/30/2018).

[29] *Le Loria*. LORIA, July 2018. URL: http://www.loria.fr (visited on 07/31/2018).