My tasks were completed in this order:

1- popular frameworks research

2- programming and mining scripts for issues and comments

3- issues reading to identify some non-crashing/model-affecting defects and keywords

4- detect how many time a bugfix is called to determine if the bug is important

4.1- find a tool that can achieve it. Python seems to have one, but C/C++ not.

4.2- code the C++ tracer

4.3- find an inserter the tracer’s call

4.4- implement a python inserter for the tracer’s call

# 1- Popular framworks research (week 1)

**Document 1:** **NN most popular libraries.docx** (<https://drive.google.com/open?id=1XH2S-7eN1wl2EYqiN9XQjaOBKwkfuyGq> )

The internship started on Monday 3rd of June. It was the beginning of the first step which was the gathering of informations about machine learning neural network most popular frameworks. After some research on the internet, most results suggested a list resembling this :

1) TensorFlow

2) Keras

3) PyTorch

4) Caffe

5) Theano

6) MXNET

7) CNTK

8) DeepLearning4J

9) Caffe2

10) Chainer

11) FastAI

And others

Reference : <https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a>

I referenced other websites the google document. I also noted each framework’s Github, bug repositories and website.

The second step of research is to find way(s) of extracting the bugs that have an impact on neural networks model. Since crashing bugs would prevent to run the deep learning models, it was important to only study non-crashing bugs.

I started to study bug repositories of TensorFlow, Keras and PyTorch trying to find a distinct aspect that separate crashing bugs and non-crashing bugs. First possibility, the repositories might have a label to indicate bugs, for example type:bug/performance for Keras and type:bug for TensorFlow. However, for a non-initiated scientist (like me), I could not be assured that the labels were used correctly and would mark down a real non-crashing bug.

Near the end of the first day, Houssem recommended me to learn how to use Python dictionaries and Github API. He wanted to mine the information about each repository and store them in json and csv files. The manual reading of Github issues should be facilitated with csv files. It would also simplify the display by accessing some needed issues.

On the next two days, I took tutorials to become familiar with Python dictionaries, sets, pandas. Emilio also taught me how to make requests to the Github REST API. I started the script\_issues.py near the end of the week. The structure of the script goes as follow :

# 2- programming and mining scripts for issues and comments (week 1 and 2)

**Document 2:** **Script\_issues.py**

From a csv containing frameworks’ information, mines each framework issues (with label or not) and saves them in a json and a csv file.

**Document 3: script\_comments.py**

From the csv containing issues, mines all comments of all issues in the csv and saves them in a json and a csv file. The comments for issues with label will be already in their own csv, since issues are already separated after using script\_issues.py.

# 3- issues reading to identify some non-crashing/model-affecting defects and keywords (mid-June 2019)

**Google Drive folder :** [**https://drive.google.com/open?id=1nSMmAo0kiAlpdIVdbZFeolyICWfQNycH**](https://drive.google.com/open?id=1nSMmAo0kiAlpdIVdbZFeolyICWfQNycH)

*I apologize for the formatting of the tables. Be mindful of the table name, as some “relevant bugs” tables are long.*

**Document 4: Manually reading all frameworks** (<https://drive.google.com/open?id=15KPcTNVlmCPgZum-dQTl-JN7oG2lMZkQ_uss6LdeA7s> )

* TensorFlow (page 1):
* Relevant: manual reading of recent relevant issues. Brief look at the repo to help finding keywords/aspects for the classification of bugs.
* Non-relevant: manual reading of recent non-relevant issues. Brief look at the repo to help finding keywords/aspects for the classification of bugs.
* Caffe (page 2):
* Relevant:
  + Legitimate issues: manual reading of recent relevant issues. Brief look at the repo to help finding keywords/aspects for the classification of bugs.
  + [label:bug] Exhaustive manual reading of Caffe issues with label:bug. The “Our Notes” columns contains Emilio’s opinion of the impact of bugs, which is a better indicator than the grade indicator in the “Issue title” column. From #3254, “Our Notes” columns’ content is an estimation of the bug impact. #297 and #284 are attempts to classify issues from Deep learning stage.
* Non-relevant (page 8): One example of crashing bug issues (crashing issue). Three possible examples of PR that probably have a minor impact on models.
* PR: is not a list of PRs, but a list of keywords.
* Sonnet (page 10):
* Relevant: exhaustive reading of all closed issues. Emilio’s notes in “Our Notes” column.
* Non-relevant: one non-relevant issue that was confusing. No Emilio notes.
* PR: No Emilio notes. I started looking at Sonnet’s PRs, but I did not continue.
* Swift for TensorFlow (page 12):
* Relevant: exhaustive reading of all closed issues. No Emilio notes.
* Non-relevant: non-relevant issues that were confusing. No Emilio notes.
* PR: empty.

**Document 5: manually reading Keras** (<https://drive.google.com/open?id=1ZJHPlkg1C0d9IOj3f6SpoegnSaG1TbGzQBfDgl19umQ> )

* Keras relevant issues (page 1):
* Relevant: manual reading of recent closed relevant issues. Covers a fewer number of issues than PyTorch, but probably not all, as I started to focus on Pytorch because of its better version’s documentation. No Emilio notes.

**Document 6: Manually reading PyTorch** (<https://drive.google.com/open?id=1m-pJxy1R00Gm4Vvi2lHc6bksKjqL8fmdVxOiaKfG-qo> )

* This document contains commit numbers for almost all issues/commits noted. The version number is also noted for those that was easier to retrace (mostly table of PyTorch relevant issues.
* PyTorch relevant issues (page 1):
* Relevant: manual reading of recent closed relevant issues. Covers a good number of issues, but probably not all, as I started using keywords. No Emilio notes.
* PyTorch keywords issues (page 4):
* Relevant: manual reading of numerous issues found using grep and gitlog with keyword “bug” (and possibly “fix” and “bugfix” too…). I would read a certain number of issues, then skip a number of issues to “randomize” the reading. No Emilio notes.
* PyTorch files history, suggestion from 9th July meeting (page 10):
* Relevant: exhaustive reading of changes history for conv.py, batchnorm.py, maxpooling.py, pixelshuffle.py and pooling.py. No interesting results for maxpooling and pooling. No Emilio notes, but he said #12952 previous commit causes a crash.

# 4- Version documentation

**Google drive folder :** [**https://drive.google.com/open?id=1lOnB1Av7gzzz6vXt\_XXx6iiXVIRLYiZq**](https://drive.google.com/open?id=1lOnB1Av7gzzz6vXt_XXx6iiXVIRLYiZq)

**Document 7:** **releases version support TensorFlow** (<https://docs.google.com/document/d/19T5njgSxdc74wnznbd3BssWKeVLZNSwq2WlBhU26PQc/edit>)

1) TensorFlow versions’ compatibility (gpu) (page 1): dependencies’ version for each TensorFlow gpu version. The table’s purpose is to know which dependencies are needed to install a buggy version.

2) whl packages for each Python version compatible with TensorFlow 1.13 (page 2): whl packages are easier for the installation ofl the buggy version. I think this table is also for TF 1.14 …

3) whl packages for each Python version compatible with TensorFlow 1.14 (page 3): whl packages are easier for the installation ofl the buggy version.

4) whl packages for each Python version compatible with TensorFlow 1.12 (page 6): whl packages are easier for the installation ofl the buggy version.

5) whl packages for each Python version compatible with TensorFlow 1.11 (page 8): whl packages are easier for the installation ofl the buggy version.

6) whl packages for each Python version compatible with TensorFlow 1.10 (page 11): whl packages are easier for the installation ofl the buggy version.

7) whl packages for each Python version compatible with TensorFlow 1.9 (page 13): whl packages are easier for the installation ofl the buggy version.

8) whl packages for each Python version compatible with TensorFlow 1.8 (page 15): whl packages are easier for the installation ofl the buggy version.

The tables for version 1.7 and earlier are not present because the study focuses on bugs corrected from year 2016 and after.

**Document 8: releases version support PyTorch, Caffe and Theano (**<https://docs.google.com/document/d/13JBxRsZd4wkD4ep2BjeJ_Cz8v081srwaAgvoNFsGWvs/edit>**)**

1) PyTorch versions’ compatibility (page 1): dependencies’ version for each PyTorch version. The table’s purpose is to know which dependencies are needed to install a buggy version.

2) whl packages for each Python version compatible with various PyTorch versions (page 2): whl packages are easier for the installation ofl the buggy version. NOTE: The links are for CUDA 7.5

3) Caffe versions’ compatibility (page 4): dependencies’ version for each PyTorch version. The table’s purpose is to know which dependencies are needed to install a buggy version.

4) whl packages for each Python version compatible with various Caffe versions (page 4): whl packages are easier for the installation ofl the buggy version.

5) Theano versions’ compatibility (page 5): dependencies’ version for each PyTorch version. The table’s purpose is to know which dependencies are needed to install a buggy version.

6) whl packages for each Python version compatible with various Theano versions (page 5): whl packages are easier for the installation ofl the buggy version.

**Document 9: meeting notes (**[**https://drive.google.com/open?id=1QKWxlSCWA1x8Q611SBamsuXsP\_\_ZtayrkmOCj6AE\_eE**](https://drive.google.com/open?id=1QKWxlSCWA1x8Q611SBamsuXsP__ZtayrkmOCj6AE_eE) **)**

Produced by Emilio. Summarizes all subjects for 17th June 2019, 24th June 2019, 1st July 2019 and 16th July 2019 meetings. A diagram at page 4 describes the workflow used for the research process of the study

# 4- detect how many times a bugfix is called to determine if the bug is important

# 4.1- find a tool that can achieve it. Python seems to have one, but C/C++ not.

# 4.2- code the C++ tracer

# 4.3- find an inserter the tracer’s call

# 4.4- implement a python inserter for the tracer’s call

Main obstacles:

* Machine performance is not sufficient (speed and ram)
* Planning problems. Emilio is not always at the lab. New tasks appeared during work, which required more time. A lot of time was spent developing tools to help the project workflow.