ShieldNet: A Deep Learning approach to DoS and malicious traffic detection

by Pascoli Massimiliano
AAU id: 12138922
e-mail: mapascoli@edu.aau.at
date: 20/02/2023

Motivation

Internet is undoubtedly a useful tool that grants almost endless possibilities in the realm of computing. It is so powerful, in fact, that represents an opportunity, but as all powerful inventions, also a potential danger. Since Internet infrastructure is now well developed and wide-spread everyone can access services online at a fairly low price and effort. A consequence of that, is the infrastructure being crowded with more and more devices that offer mentioned services, some of which are critical, real-time the assets managed must be always available and accessible. It is not wrong to state that many devices connected, given the number, are vulnerable to cyberattacks. The latter are certainly growing in frequency and intensity because of the connected asset value stored online increasing too.

If for intrusion detection, exploits and misconfiguration of such devices there are easy and applicable solutions (stronger crypto algorithms and passwords, firewalls, keep systems up-to-date, utilizing only trusted software, ...), for DoS (denial of service) and DDoS (distributed DoS) attacks there are not. This is because DoS type attacks are technically always feasible: if there is a connection with other machines, then the attack is applicable and it doesn't help that these attacks are conceptually quite simple to set up.

In its simplest form, a DoS attack aims to exhaust network bandwidth or connection management capabilities of the victim, so that no one can connect to the victim and its services. It is important to note that a DoS attack does not undermine confidentiality, nor integrity of the asset, accessibility only. Although, some confidentiality attacks can, indeed, generate DoS collateral traffic (e.g. online password brute force).

Fortunately, there are some "classical" statistical ways to mitigate these attacks. All of them are based on user-defined thresholds and the knowledge of average dynamics in the network. These methods are based on Shannon entropy estimation or correlation to create a network representation and are based on features like who is connected (source IP). Already using common DoS or DDoS tools freely available (e.g. LOIC and HOIC resp.) an attacker can manipulate IP addresses, invalidating an on-paper accurate representation.

But what to do if dynamics change a lot or it is not possible to define an "average" network behaviour? A deep learning approach can help, since it is flexible, adaptable, updatable and independent from network "average" representation. They can learn complex attack patterns to master the vast taxonomy of DoS attacks.

In order to accomplish the task, this project presents the approach based on multi-layer perceptron (MLP) models that can classify a connection flow (data belonging to a specific connection i.e. metadata regarding multiple internet packets in aggregated form) as *benign* or *DoS* reaching state-of-the-art performances (and above). Moreover, some model presented is able to distinguish between 11 different attack techniques: this could be very informative in order to adopt ad hoc countermeasures not discussed in this paper. During the realization of this project some exploration was made, concerning:

- impact of different activation functions;
- model weight initialization schemas;
- feature reduction with ANOVA (Analysis of Variance) and MI (Mutual information);
- autoencoders.

Data

We want to solve two variant of one classification problem. Each (bidirectional) connection flow defined by

$$\langle \text{source\_ip}, \text{source\_port}, \text{destination\_ip}, \text{destination\_port}, \text{protocol} \rangle$$

can be used for:

- binary classification task (BCT) in *benign* or *DoS*;
- multi-class classification task (MCCT) in *benign* or one particular *DoS technique*.

Two different datasets were used for training in MCCT and BCT settings:

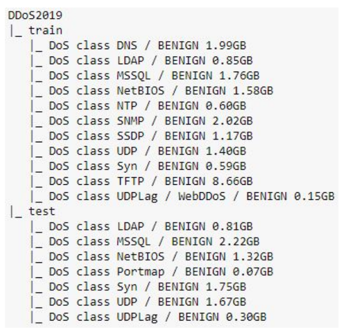
- MCCT CIC-DDoS2019;
- BCT this Kaggle dataset (file "ddos_balanced.csv" only), which is a balanced mixture of data coming from other very popular IDS datasets, in particular:

    - CSE-CIC-IDS2018-AWS;
    - CICIDS2017;
    - CIC DoS dataset(2016).

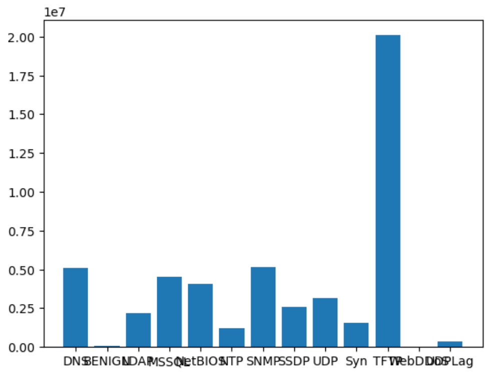
The data used was generated with the same learning task in mind, as can be seen in the reference pages: it was recorded on a test network diverse enough to include several possible attack devices and victim devices. The raw packet data was recorded using libpcap (OS independent library that is available for use in many programming languages) available with *tcpdump* utility and then, using CICFlowMeter some aggregated metrics were computed (e.g. flow duration, average packet length, packets per second, ...). To keep this section brief, please refer to Table 3 in this page to know what metrics this tool can calculate given raw packets in pcap standard and what each feature represents. These measurements are present in both datasets, except for different naming conventions.

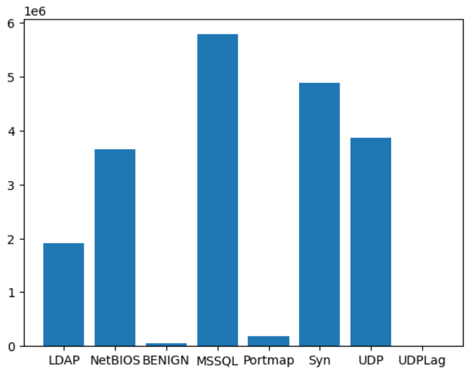
Properties and structure of BCT & MCCT datasets

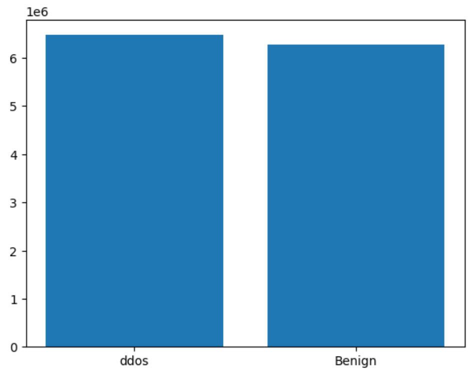
Both datasets are far from usable as are downloaded from the repositories, especially for MCCT. The distribution of classes and dataset structure is as shown below:



```
DDoS2019
|_ train
   |_ DoS class DNS / BENIGN 1.99GB
   |_ DoS class LDAP / BENIGN 0.85GB
   |_ DoS class MSSQL / BENIGN 1.76GB
   |_ DoS class NetBIOS / BENIGN 1.58GB
   |_ DoS class NTP / BENIGN 0.60GB
   |_ DoS class SNMP / BENIGN 2.02GB
   |_ DoS class SSDP / BENIGN 1.17GB
   |_ DoS class UDP / BENIGN 1.40GB
   |_ DoS class Syn / BENIGN 0.59GB
   |_ DoS class TFTP / BENIGN 8.66GB
   |_ DoS class UDPLag / WebDDoS / BENIGN 0.15GB
|_ test
   |_ DoS class LDAP / BENIGN 0.81GB
   |_ DoS class MSSQL / BENIGN 2.22GB
   |_ DoS class NetBIOS / BENIGN 1.32GB
   |_ DoS class Portmap / BENIGN 0.07GB
   |_ DoS class Syn / BENIGN 1.75GB
   |_ DoS class UDP / BENIGN 1.67GB
   |_ DoS class UDPLag / BENIGN 0.30GB
```







CIC-DDoS2019 (first row):

- *BENIGN* class is not separated in its own file, but it is mixed with all the other classes in every csv;

- not all classes present in the training part are present also in the test part (and vice-versa): the training/test split suggestion is useless for our task, might as well ignore it completely;
- classes are far from balanced;
- the size of every csv listed ranges from 0.1GB to 8.7GB given a total of 28.9GB: impossible to load all dataset at one or only training/test part to have a more or less complete view;
- *WebDDoS* and *Portmap* traffic samples are almost non-existent compared to *BENIGN* class, and *Portscan* is technically not a DoS technique, but a service and port probing tool...

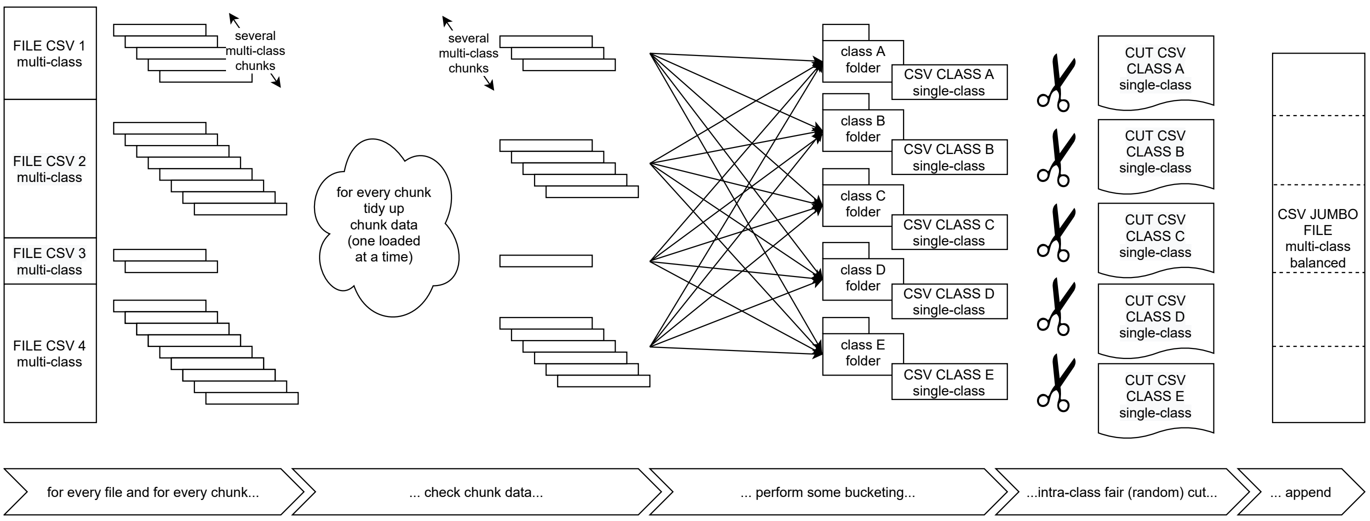
Kaggle dataset (second row):

- one file only, almost balanced, but prohibitive size of 6.32GB.

Furthermore, **both dataset** contain:

- NaN/missing values;
- non numeric values e.g. IP addresses;
- features are naturally out of scale with respect of each other (e.g. total number of packets in a flow - possibly billions; and packets per second - usually 80, 22, ... or packets per second...).

To deal with all (many) problems listed above, the solution is summarized in the image below:


FILE CSV 1 multi-class
FILE CSV 2 multi-class
FILE CSV 3 multi-class
FILE CSV 4 multi-class
several multi-class chunks
for every chunk tidy up chunk data (one loaded at a time)
several multi-class chunks
class A folder
CSV CLASS A single-class
class B folder
CSV CLASS B single-class
class C folder
CSV CLASS C single-class
class D folder
CSV CLASS D single-class
class E folder
CSV CLASS E single-class
CUT CSV CLASS A single-class
CUT CSV CLASS B single-class
CUT CSV CLASS C single-class
CUT CSV CLASS D single-class
CUT CSV CLASS E single-class
CSV JUMBO FILE multi-class balanced
for every file and for every chunk...
... check chunk data...
... perform some bucketing...
...intra-class fair (random) cut...
... append

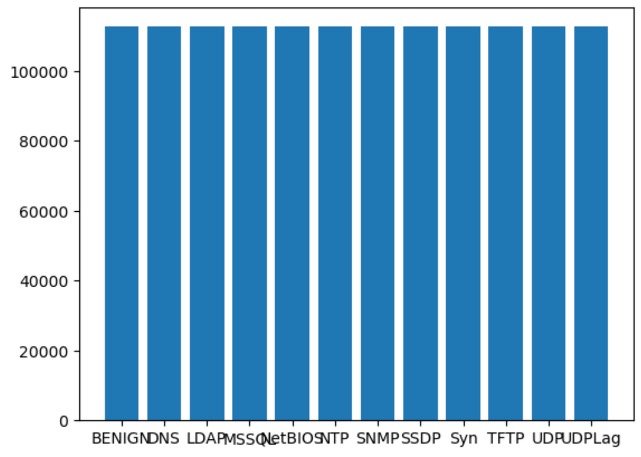
Reading of files is not done all at once, but in chunks. Every chunk is processed deleting rows that contain missing values (since we have more than sufficient samples, we don't need to fill the missing values - e.g. with mean plus noise, we can do away with the samples altogether). At the same time we drop useless column or useless labels such as *Portscan* and *WebDDoS*.

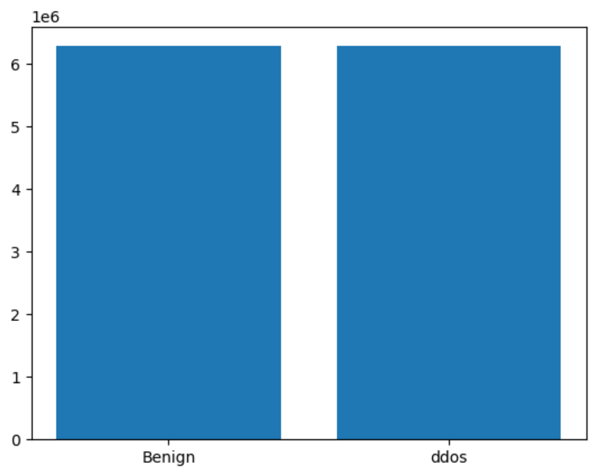
In order to rebalance labels, we cut down all classes to the number of samples the least numerous class has, choosing randomly from overrepresented labels.

To deal with out of scale features we will use a standard scaler before training.

After all the data manipulation, we are left with balanced MCCT dataset in a single 0.43GB file that we can easily read all at once and one 5.12GB BCT file. For BCT, the exploration of models and hyperparameters will be done on a subset of the data generated in a similar fashion as the 5.12GB csv.

Below the classes distribution in detail:





Literature

This section summarizes the most interesting works on the topic that I could find online and explains some thoughts that I had when choosing what to implement to solve the learning task.

It seems that in literature two different approaches are mainly used to deal with this learning task:

- classify **every** incoming packet to *DoS* or *benign*
- classify **aggregated data of a flow** (flow = multiple packets) in the two classes already mentioned

As for the first approach, we would like to mention Paper#1a and Paper#2a. Both using CNNs on single packets encoded as images (byte to byte greyscale and bit to bit b/w resp.). Results are impressive, with above 99% accuracy on BCT. Pater#2a also shows that relatively simple DL models can compete with (and are superior to) other ML approaches (e.g. SVMs are mentioned).

For what the second approach is concerned, that is also the one we adopted in this project, we would like to comment on four paper in particular. Paper#1b is interesting because, in order to deal with a really unbalanced dataset, Synthetic Minority Oversampling Technique (SMOTE - the implementation of SMOTE and the functioning presented originally here) was used, combined with LSTMs. SMOTE becomes useful when dealing with unbalanced learning: instead of classical oversampling, new "synthetic" samples are generated introducing noise or averaging the already existing samples. Paper#2b using Graph Attention Networks and extremely recent (16th Aug 2022). The researchers viewed traffic flows as a graph with metrics on the edges that represent the communication flow. This representation is then used in GNNs. Paper#3b that compares different ML approaches and draws current state of DoS detection techniques providing also numerous datasets to use for the task. It was useful in the assessment at the beginning of the project because gives a panoramic of the best datasets available for our needs. Finally Paper#4b makes use of autoencoders. These powerful architectures were tried in one of the attempts made and produced very good (state of the art) results with an extremely simple model. All approaches in papers Paper#Xb reach an accuracy around 98-99%, but make use of relatively large and complex models: we would like to simplify as much as possible (simple MLP classifiers) without taking away state-of-the-art performances (we reached 99.9% accuracy in BCT).

Why the second approach?

This project wants to be the core part of a complete and deployable proof of concept application we are developing (Small Project in Cybersecurity course). This means that the model produced should be able to work online at line rate, not on packet dumps offline. A computationally lightweight product is therefore required to not kill packets throughput. Using the first approach would cause an evaluation of a model at every packet: this is too much e.g. for a router. Another important reason is that managing flow information is much easier: million of packets can be aggregated and distilled into 80 features without keeping them saved in memory (we can do away with the content of the packet entirely). When a packet arrives, it is only required to update only counters, means, variances, ... This can be done efficiently with one-pass algorithms.

There is a downside: in order to use the model in a real world scenario, it is needed, first, to aggregate some (many) packets to compute meaningful features values and only then, apply the model. With the other approach it would be possible to immediately distinguish a packet as *DoS* or *non-DoS*.

Implementation & results evaluation

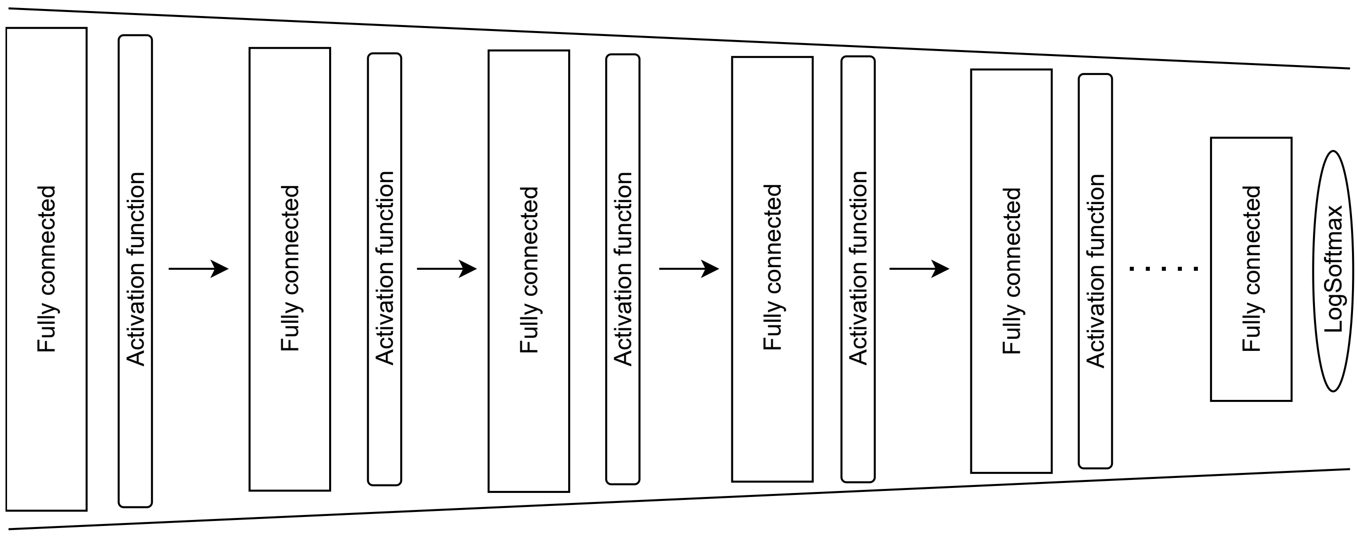
In this section are presented the structure of most important models created in detail and only briefly the ones that allowed exploration of the problem at first. In any case, for an in-depth view, code is available.

The development was done entirely on my local machine. The following are some characteristics of the system:

Hw	Sw
CPU: Intel i7-4790K	ENV: JupyterNotebook in VisualStudioCode with Python3.9 and Pytorch
RAM: 20GB DDR3	SO: Windows10 Home 19045.2486
GPU: NVIDIA GeForce GTX 970 4GB GDDR5	

Here are some general "implementation fix points" valid for every model produced, from exploration phase to later stages:

- as discussed in previous chapters, data was rescaled using a *StandardScaler*, since feature are very diverse in magnitude one from each other. The scaler was fit on the training set attached to each model, each time;
- this is a classification task, so at the very end of each model a *LogSoftmax* layer is used (same concept of standard *Softmax*, but numerically more stable);
- to compare *LogSoftmax* output of a model with a sample's one-hot-represented ground truth, *CrossEntropyLoss* was used (this is suitable both for BCT and MCCT);
- the available data was always divided following the ratio 75% / 25% for training set and test set respectively. When validation set was also needed, the training set was further split following the 75/25 formula (training set and validation set resp.);
- for the performance assessment of each model, were used standard performance metrics for classification i.e. *Recall*, *Precision*, *F1-score*, *Accuracy*, *Per-Class Accuracy* in their BCT and MCCT variants;
- all models are following the structure (with due changes where needed) in the image below



Exploration

We started with MCCT. The first very few models were very small, plain and simple Multi-Layer Perceptrons with maximum 128 neurons per layer, interleaved with some sort of activation function. These models allowed a fast training and a lot of exploration with hyperparameters and what activation function to use.

Based on epoch loss values trend it was possible to estimate a good learning rate for a solid convergence and the best activation functions (it was noted that changing activation functions hugely impacted loss magnitude so were kept the ones that minimized it) keeping batch size the same (200). Activation functions tried ranged from Sigmoid to ReLU, GELU, ELU, LeakyReLU, SeLU and, most importantly, Tanh. Tanh is a very sharp activation function that worked well: took initial 6000+ loss to ~3000. Also mixtures were attempted, but didn't work well at this stage.

In general, the slope of the loss stabilized to "flat" in already 10 epochs, but the performance was very low and not acceptable (0% accuracy per some classes). Class *Benign* was the most accurate, this is a sign that the BCT might be very easy because of a clear distinction between normal and abnormal traffic flows.

ANOVA, MI, Autoencoders

After the first unsuccessful attempts on the MCCT, we moved on BCT. A slightly larger model (`512, ReLU, 256, ReLU, 128, Tanh, 64, Tanh, 2, LogSoftmax`) was capable of reaching state of the art (and above) performance:

```
Precision: tensor(0.9968)
Recall: tensor(0.9961)
F1-score: tensor(0.9965)
Overall Accuracy: tensor(0.9965)
Per class accuracy:  [0.9968386848929155, 0.9961056200620542]
```

This model was classifying on feature-reduced data. In order to compute the reduction of the features, ANOVA *f-classif* was used (this is to account for possible linear relations between features).

As suggested during the presentation, *mutual-info* (MI) was also tried (for non-linear relations) and the results are reported below:

```

 s
```

MI takes much longer to compute, compared to ANOVA f-score: this is probably because it is based on k-nearest neighbours distances. Performance is quite similar nonetheless, starting from different selected features, obviously.

To deal with BCT, also autoencoders were used. A small autoencoder, coupled with the same classifier structure of ANOVA and MI approaches, is capable to reach comparable performance. The drawback is that training is longer and performance is technically less than state of the art:

```
 Per class accuracy:  [0.9851537051709925, 0.9730514261952932]
```

Definitive models

In this section are presented the best models realized for MCCT and BCT. The classifier structure is the same for both tasks: (`512, ReLU, 256, ReLU, 128, Tanh, 64, Tanh, 2, LogSoftmax`). What changed in the training process was the inclusion of weight initialization and learning rate scheduling. A summary of all the hyperparameters is given in the following.

MCCT v1

- batch size = 100;
- `torch.nn.init.xavier_uniform_` weight initialization (best among default, `xavier_normal_`, `xavier_uniform_`, `kaiming_uniform_`, `kaiming_normal_`);
- structure reported below;
- 50 epochs;

- learning rate = 0.001;
- Adam optimizer with CrossEntropyLoss;
- learning rate optimizer `StepLR` with step_size = 10, gamma = 0.1;
- a rudimentary early stopping mechanism

```
Precision:  tensor(0.7203)
Recall:  tensor(0.6648)
F1-score:  tensor(0.6208)
Overall Accuracy:  tensor(0.6648)
Per class accuracy:  [0.3305182203694387, 0.999355150733516, 0.8797976792621244,
0.7496016712105654, 0.05497496189854126, 0.9750008839857148, 0.9800937237153042,
0.273768573353665, 0.8943164471582236, 0.47682616494931374, 0.9968313046977165,
0.36667516294650987]
```

For what BCT is concerned, since we worked on a subset of all the 12 millions of samples (2 million at first in exploration phase and 5 millions for the definitive model), we would like to know what is the performance is on all the 12 millions samples: this lead to surprising results.

Conclusions & future steps

A report should be a 6-8 pages long document structured as a research paper, as shown below. You may modify this structure, but the suggested list is a good starting point.

Motivation

describe the problem you are solving, explain why is it important, and provide a short summary of your methods and obtained results.

Data

describe the data you are using in the project and provide examples. what kind of data you are using, where does it come from? what are the properties of your data, like balance, missing values, scale, etc., and what are you going to do about it? what kind of preprocessing, filtering, augmentation, or other manipulations are you applying in the project?

Theoretical part

literature overview: which models from the literature are suitable for this problem given your data and why? which algorithms can be used to train them? how these models are related to your hypothesis? present an overview of your approach: what steps are executed in the learning pipeline? which hyperparameters of the learning algorithm are available and how they can influence the results? You must show that you applied methods considered during the semester to the selected problem. Feel free to include any figures or tables helping to describe your method and compare it with others. If you use information from other sources, please cite it properly.

Implementation

how did you implement your approach? which tools were used? show interesting snippets or present your algorithms

Evaluation

present and visualize results of your evaluation, explain what do your results mean, why was your approach successful, why not? compare it to some baseline either implemented yourself or from the literature, blogs, Kaggle, etc.

Conclusions

how can you evaluate the results of your work? what would you recommend for future steps?

Supplementary materials: The report must be submitted as an archive comprising all code and artifacts (data, custom python modules, images, videos, etc.) required for its correct representation, evaluation, and exemplification of your work. If data is too large for a submission, please provide a link where this data can be downloaded from.