Presentation of ShieldNet

by Pascoli Massimiliano
AAU id: 12138922
e-mail: mapascoli@edu.aau.at
date: 12/01/2022

The learning task

ShieldNet aspires to be an effective and efficient tool to detect Denial of Service (DoS) attacks, that would allow a human/machine to take countermeasures manually/automatically with low responsive times. Since the problem is not easy to resolve due to the wide taxonomy of DoS attack techniques, it is beneficial to specify the tuple $\langle Task\_to\_perform, Performance\_measure, Experience\_data \rangle$ that characterize the learning task for the machine learning problem I am going to face with.

⚠ **WARNING**

Please note that every component of the learning task is split in two, this is to account for the **multiple** produced models: there were some technical difficulties with the multi-calss classifcation at first (attempted many different approaches) that got resolved in the end, exploiting the structure and workflow of the binary classification models produced during exploration (this was also a great opportunity to experiment with architectures, hyperparameters, data reduction techniques, ...).

Task to perform

DoS detection via packet flows aggregated metainformation, that is:

- the system must be able to distinguish if the connections flows (traditionally identified by tuple $\langle \mathbf{source\_ip}, \mathbf{source\_port}, \mathbf{destination\_ip}, \mathbf{destination\_port}, \mathbf{protocol} \rangle$) belong to a DoS attack or are just benign traffic. If the flow is malign, possibly know what kind of attack it is (multi-class classification);
    - 12 classes present in the datasets used after the initial clean-up, representing 11 different DoS attacks techniques and the normal traffic class
- there is an easier binary classification variant of the same problem which consists only in classifying between malicious and benign traffic flows starting from the same aggregated data mentioned above.

Performance measure

In general the system should be trained to minimize the number of miss-classifications, both in the binary and multi-class variant. It is particularly important a stong and correct discrimination between benign traffic and all the other malicious classes. If a flow is classified as malign, it should really be, it is less important that the attack technique is recognized correctly (nevertheless a correct classification can be very informative to apply further, more precise, countermeasures).
In the table below are shown the specific metrics (in PyTorch) used to evaluate the goodness of a model after training:

BINARY	**MULTI**
Per-class Accuracy	Per-class Accuracy
Binary precision	Multiclass precision
Binary recall	Multiclass recall
Binary F1-score	Multiclass F1-score
Binary Overall Accuracy	Multiclass Overall Accuracy

⚠ **WARNING**

Classes will be balanced after the preliminary data analysis. Technically I could have used accuracy as my only meaningful metric, but the other metrics are valid also with unbalanced problems.

Reminder: metrics

		Ground truth	
		0	1
Class prediction	0	True Negative	False Negative
	1	False Positive	True Positive

Metric	Meaning
$\text{Precision or Positive Predicted Value (PPV)} = \frac{TP}{TP + FP}$	Out of all *True/Positive/1* predicted values (i.e. class prediction *True/Positive/1*), how many of them were

Metric	Meaning
	really *True/Positive/1* in %?
$\text{Recall or True Positive Rate (TPR), Sensitivity, Hit Rate} = \frac{TP}{TP+FN}$	Out of all samples that were really *True/Positive/1* (i.e. ground truth *True/Positive/1*), how many were predicted as *True/Positive/1* in %?
$\text{F1-score} = \frac{2}{\frac{1}{TPR}+\frac{1}{PPV}}$	Harmonic mean of Precision and Recall
$\text{Accuracy (ACC)} = \frac{TP+TN}{TP+TN+FP+FN}$	Out of all samples, how many were correctly classified in %?
Per-class Accuracy	Correctly predicted for class i (i.e. prediction equals ground truth) over total samples with

Metric	Meaning
	same ground truth class i

Experience data

In order to train the models, two different datasets were used:

- for the multi-class classification problem **CIC-DDoS2019** was used;
- for the binary classification problem **this** dataset was used (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)**.

Both datasets contain information of many generated internet traffic flows, labeled with *benign*, a specific attack name or general *DoS*. The data I used was generated with the same learning task in mind: it was recorded on a test network diverse enough to include several possible attack devices and victim devices. The data was recorded capturing raw packets 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 (please refer to **this page** to know what metrics this tool can calculate given raw packets in pcap standard). Around 80 of these metrics were chosen to be present in the datasets and not all of them are useful for my goal, nor for every model proposed in the Implementation section.

⚠ **WARNING**

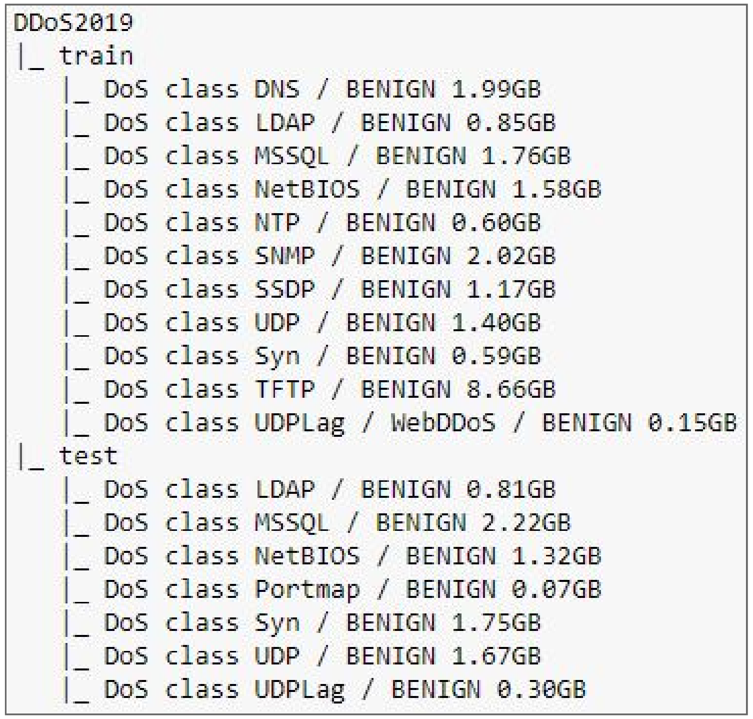
Datasets contain **flow data**: around 80 metrics for every flow were computed from many (millions, bllions, ...) internet packets belonging to the same flow! Technically, a single flow is directional ($flow(A \rightarrow B) \neq flow(B \rightarrow A)$), but in both datasets are considered bidirectional!

Examining the datasets

In this section, my goal is to understand the data, its organization and representation, in order to get ready for learning.
In the end we would like to have more or less **balanced** datasets, this is to help learning a correct inference model/function, for both the multi-class classification task (MCCT) and binary classification task (BCT). Furthermore, it is important for the purposes of training that the data we provide to the ANN is **not malformed or missing** and is **numeric**.

MCCT dataset analysis

The schematic structure of CIC-DDoS2019 dataset is the following:



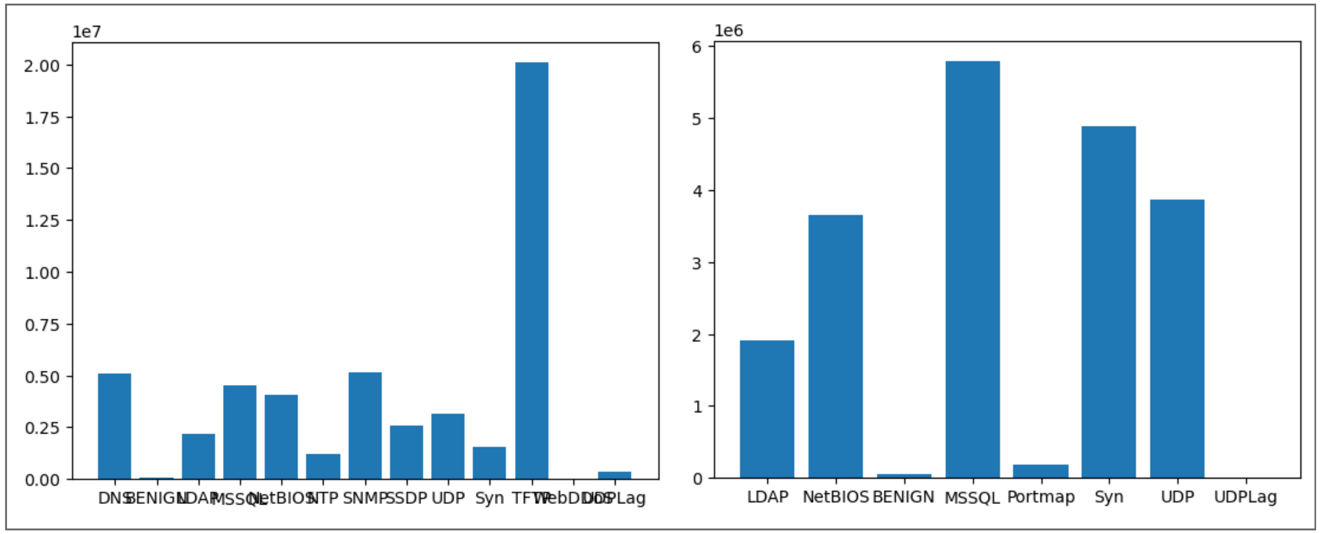
```
DDoS2019
|_ train
    |_ DoS class DNS / BENIGN 1.99GB
    |_ DoS class LDAP / BENIGN 0.85GB
    |_ DoS class MSSQL / BENIGN 1.76GB
    |_ DoS class NetBIOS / BENIGN 1.58GB
    |_ DoS class NTP / BENIGN 0.60GB
    |_ DoS class SNMP / BENIGN 2.02GB
    |_ DoS class SSDP / BENIGN 1.17GB
    |_ DoS 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
```

As you can see, the raw dataset is difficult to use for at least 4 reasons:

1. BENIGN class is not separated in its own file, but it is mixed with all the other classes in every csv;
2. 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;

3. Classes are far from balanced;
4. 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.

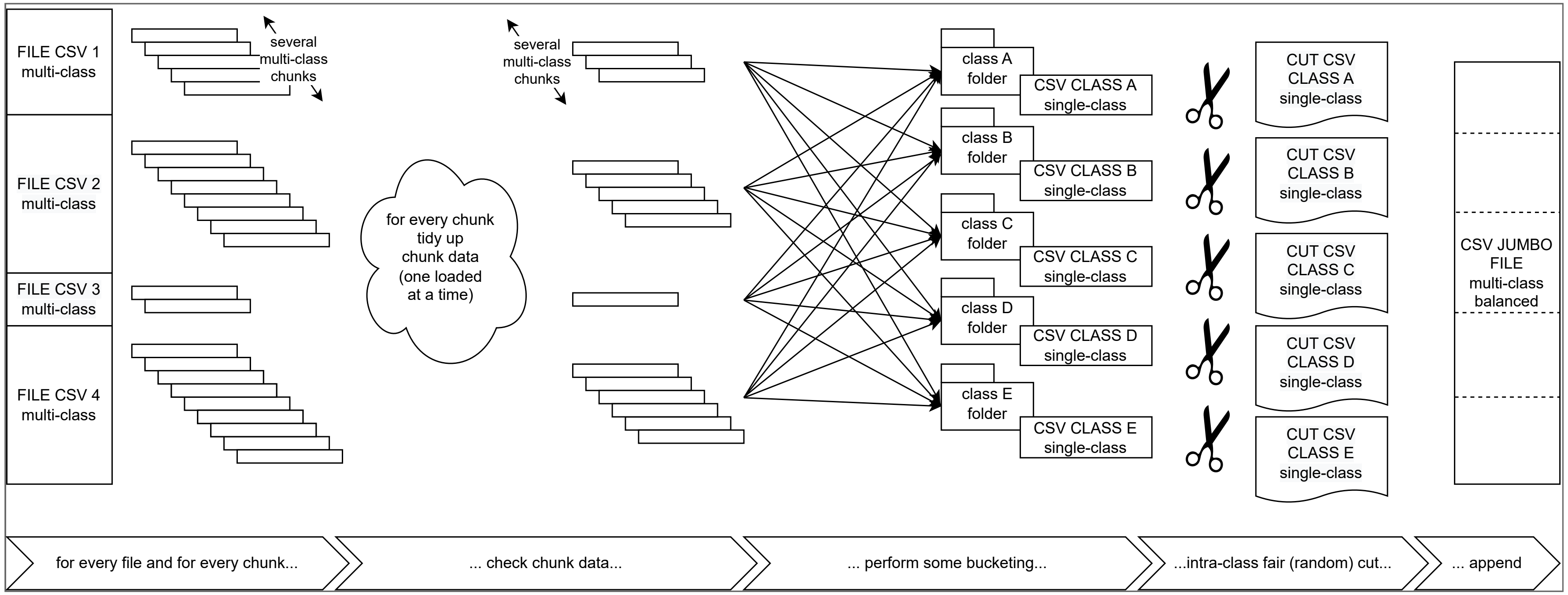
Classes distribution of CIC-DDoS2019 in detail for point 3 presented previously (training part left, test part right):



The number of samples for each class is completely out of scale with respect to others.
WebDDoS and Portmap traffic samples are almost non-existant compared to BENIGN class, and Portscan is technically not a DoS technique, but a service and port probing tool...
WHAT DO WE DO?

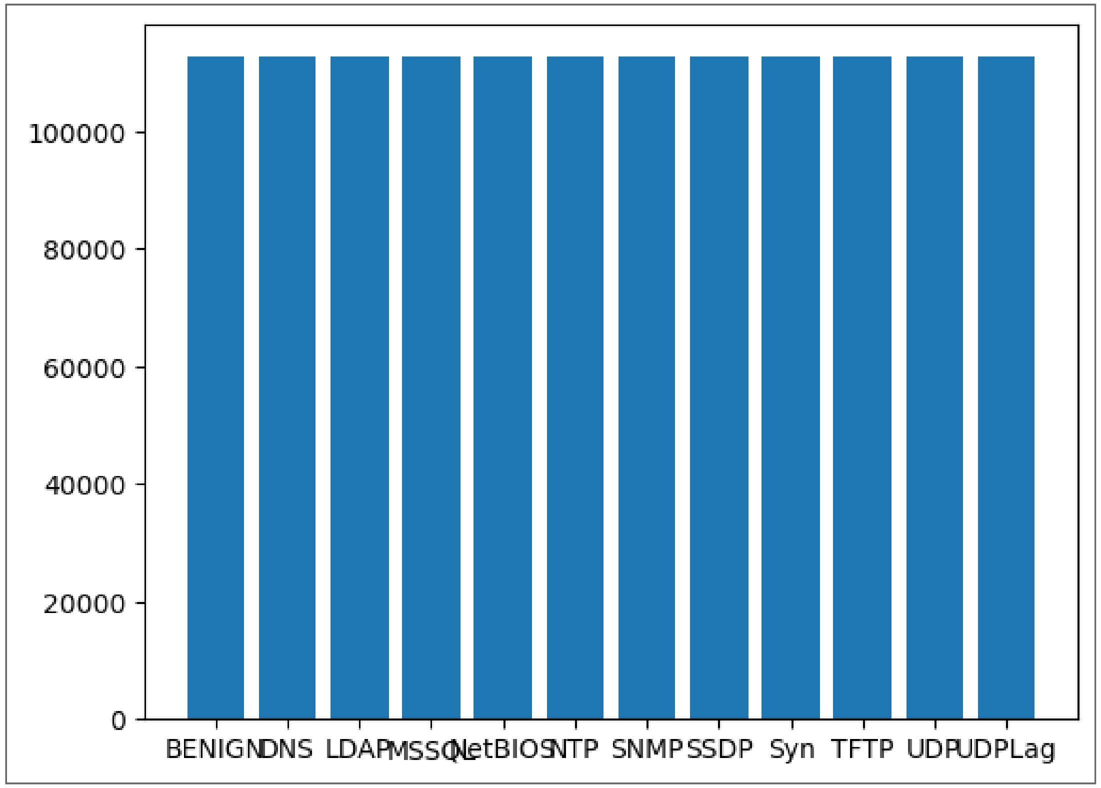
Solution to MCCT dataset

Given the dataset in the form described and since my computing power is limited, the solution implemented is to reconstruct a single, unified, balanced dataset cutting down in a **fair** manner all oversized classes, dropping Portmap and WebDDoS classes entirely. Below, a brief data transformation schema is reported:


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

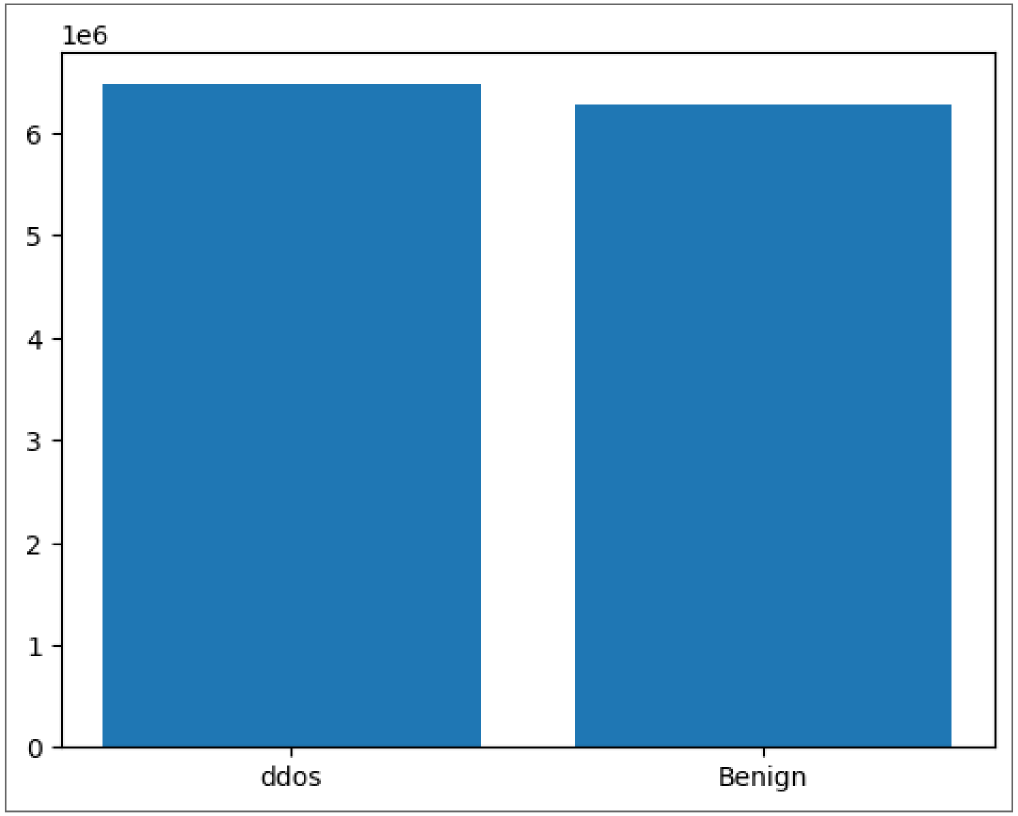
The final result is a csv file in which samples belonging to every class are cut down to the number of samples that the least numerous class has.
As a collateral product, not only the dataset has the desiderable properties described, but also is only 0.43GB. This allows me to load all data to main memory

at once as a single DataFrame (empirically main memory occupied by DataFrame object is two times the csv file size).
Every class now has 112731 samples, bringing the total number of samples to $112731 \cdot 12 = 1352772$ as shown below:

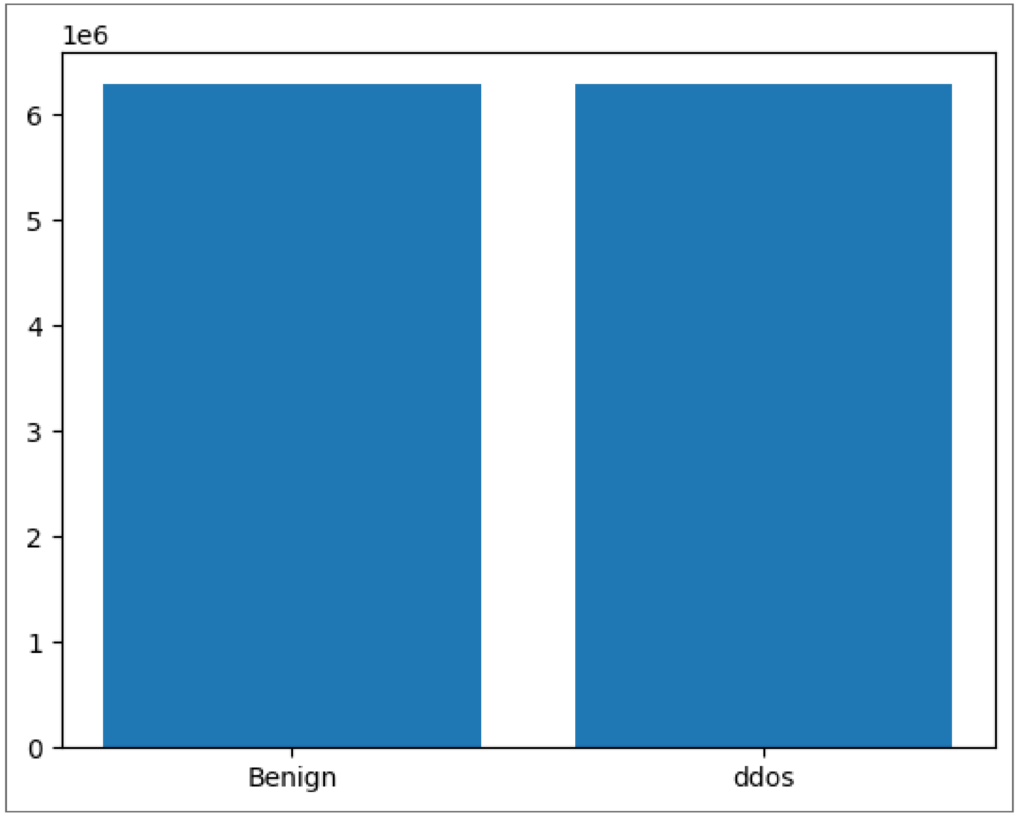


Solution to BCT dataset

Here the problem is slightly different, but the solution and the code used is exactly the same. This dataset is composed by one file only that has all the labeled flows samples I am going to use. This file contains almost the same columns as the MCCT dataset (in the result there will be the same columns exept for different naming conventions...).
It also starts pretty balanced as you can see, but its size is prohibitive (6.32GB):



After the application of my fair cut utility:



Unfortunately the resulting size is huge nontheless: 5.12GB for $6274230 \cdot 2 = 12548460$ samples. For initial exploratory tries in PyTorch I will work on a subset of this result csv (around 1000000 samples).

Literature overview

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*
    - **<u>Paper#1a</u>** with a clever approach to code a single packet to an RGB and grayscale image (considering groups of bytes in packets fields) and apply CNNs to try and do some fast classification at line rate, then comparing results between RGB and greyscale inputs. Results are impressive, in both cases with 99+% accuracy;
    - **<u>Paper#2a</u>** similar to *Paper#1a*, but using a single bit as pixel (images now in black and white). In this case accuracy is slightly lower (always above 99% anyways). Furthermore, the model is larger compared to the smallest model presented in *Paper#1a* (that is also more accurate). This paper compares results with other ML approaches (e.g. SVM) and shows that already simple DL methods implemented are far superior;

- classify **aggregated data of a flow** (flow = multiple packets) in the two classes already mentioned
    - **Paper#1b** using Synthetic Minority Oversampling Technique (SMOTE - the implementation of SMOTE and the functioning presented originally **here**) and LSTMs. This is 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). View traffic flows as a graph with metrics on the edges that represent the communication flow, then apply 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;
    - **Paper#4b** makes use of autoencoders, I tried them in one of my attempts and produced very good (state of the art) results with an extremely simple model.

Given the information presented, I decided to go with the second approach: WHY?

- The model I am going to produce should be able to work online at line rate, not on packet dumps offline (see Small Project in Cybersecurity results when ready).

This means it should be as lightweight as possible to not kill packets troughput. Using the first approach would cause an evaluation of the model at every packet: this is too much e.g. in a router.

- Managing flow information is much easier: million of packets can be "compressed" and organized into 80 features without keeping them saved in memory (especially, I can drop the content of the packet entirely). When a packet arrives, I am going to update only counters, means, variances, ... This can be done efficiently with one-pass algorithms.
- The downside is that, in order to use the model in a real world scenario, I need first to aggregate some (many) packets to compute meaningful features values and only then, apply the model. With the other approach I could immediately distinguish a packet as DoS or non-DoS.

Implementation

In this section I will present briefly what I tried during my numerous attempts. I will make a digression explaining in detail the best one found for the BCT and MCCT. For all models, a standard scaler was fit on training input data and then used first on training data, then on the test set.

$$\forall i, j \left(x\_scaled_{i,j} = \frac{x\_original_{i,j} - \mu_i}{\sigma_i} \right)$$
$$\text{where } i \text{ is a feature and } j \text{ is a sample}$$

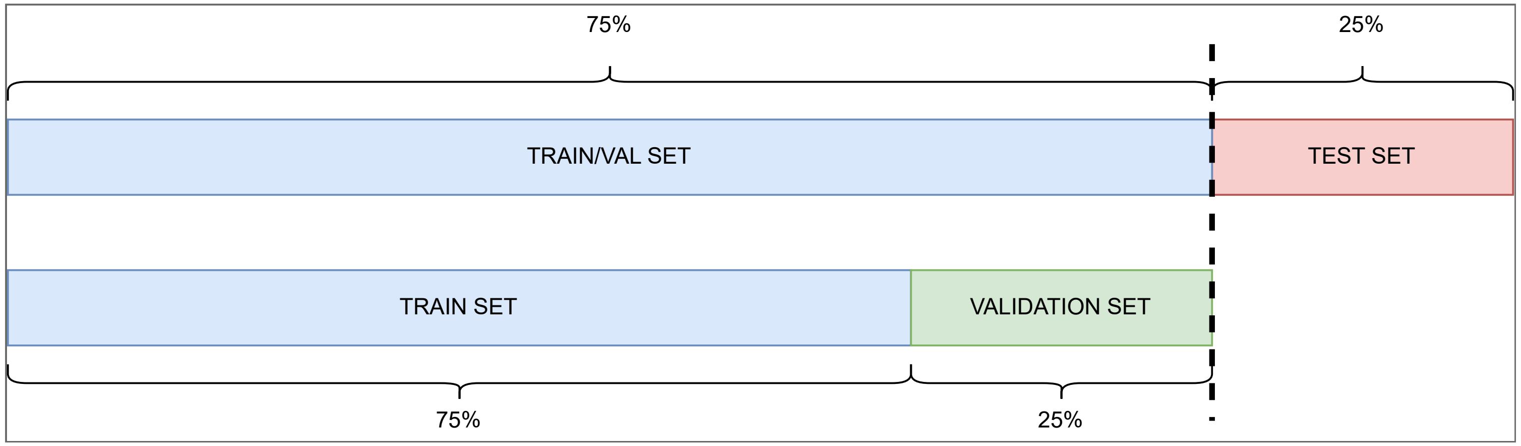
Since the task is classification and not regression, the loss function used is cross-entropy loss combined with logaritmic softmax at the end of every model.

$$L_{cross-entropy} = -\sum_{i=1}^{n} t_i \cdot \log(p_i)$$
$$\text{where } t_i \text{ is the ground truth label, } n \text{ number of classes and } p_i \text{ is the softmax probability for the } i^{th} \text{ class}$$

$$LogSoftmax(x_i) = \log\left(\frac{e^{x_i}}{\sum_j^n e^{x_j}}\right)$$
$$\text{where } x_i \text{ is a component of an n-dimensional tensor}$$

The optimizer chosen for every model is a pretty standard and popular one with default settings and learning rate (0.001): **Adam** (Adaptive Moment Estimation). The

value for the learning rate was found empirically after some initial runs on very small models to be good for convergence on the data of both MCCT and BCT datasets.
In general, datasets were divided as shown below for training, validation and test sets:

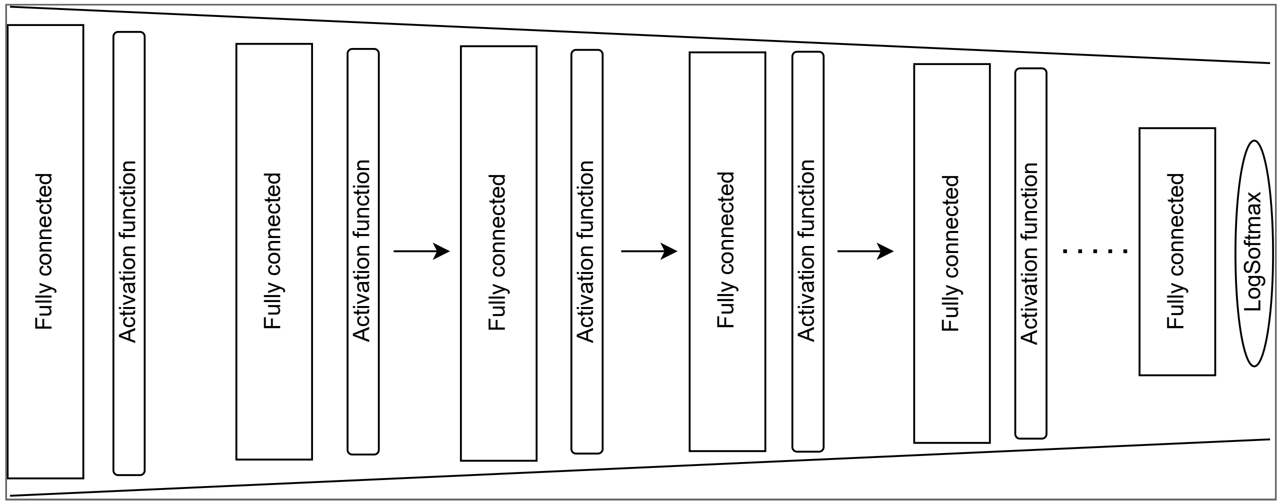


First steps (*"is convergence even possible?"*)

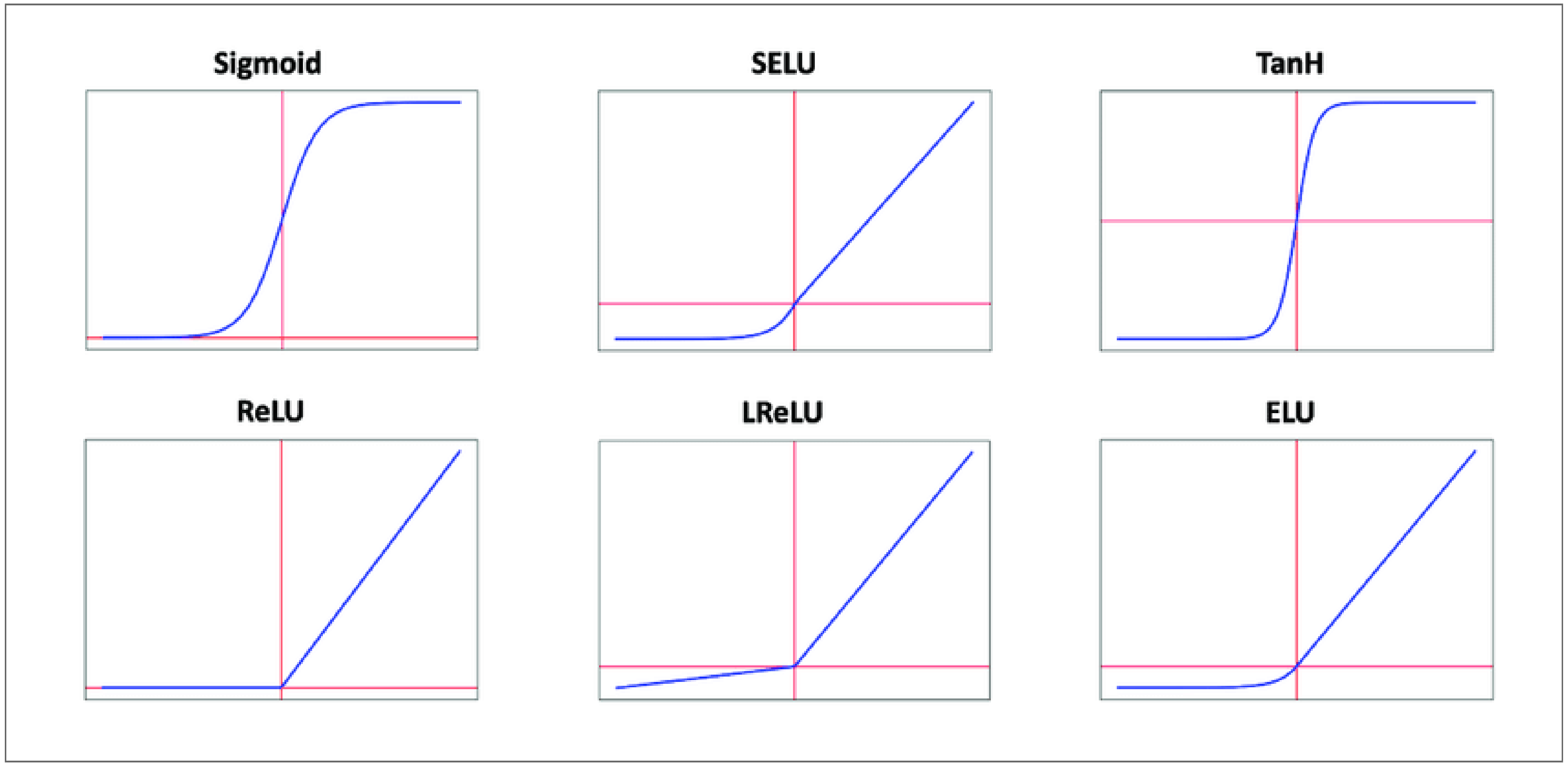
⚠ **WARNING**

The results of this stage were bad, but some important discoveries were made for what model morphology is concerned. I didn't use the validation set at all in this early stage!

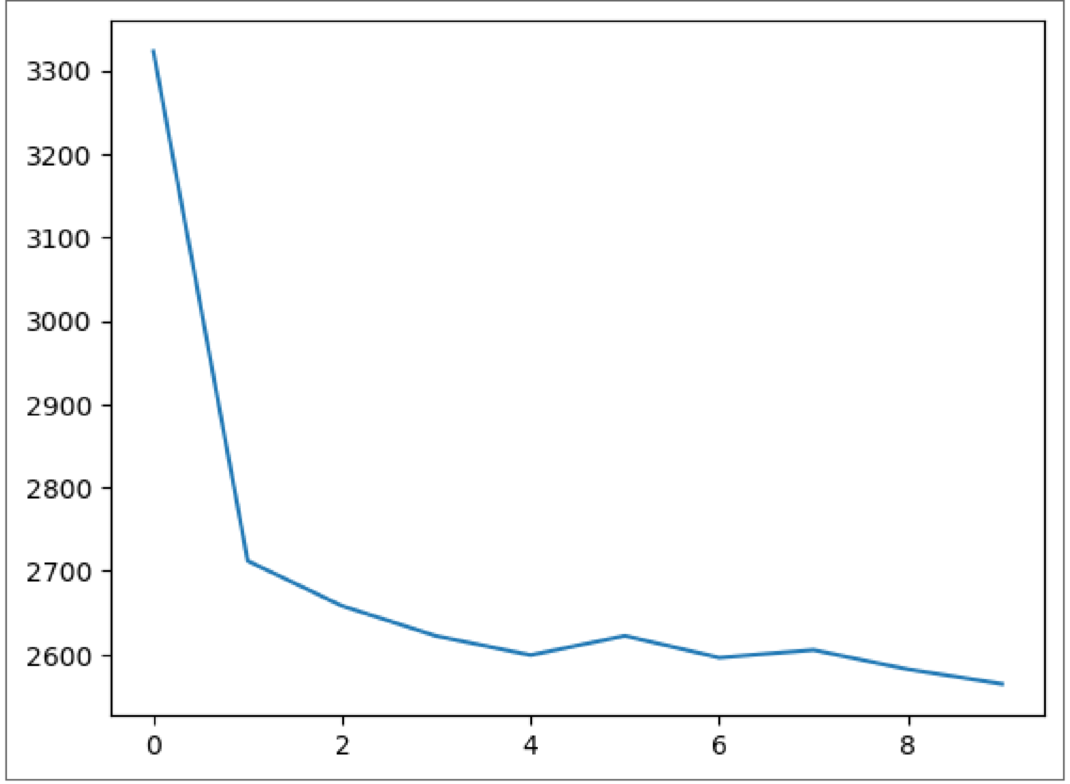
For starters, I tried to solve MCCT with really simple models that could be trained easily and in little time. Generally all models were low parameter number MLPs (i.e. Multi-layer Perceptrons with at maximum 128 neurons per layer) interleaved with some sort of activation function and, as mentioned, logarithmic softmax at the end.



Based on epoch loss values trend I could estimate a good learning rate for a solid convergence and the best activation functions (I saw that changing activation functions hugely impacted loss magnitude so I kept the one that minimized it) keeping batch size the same (200). I tried all between Sigmoid, 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.



https://www.researchgate.net/figure/Sigmoid-Scaled-Exponential-Linear-Unit-SELU-Hyperbolic-tangent-TanH-Rectifier_fig1_350125835 The plot below shows loss value using Tanh per epoch:



The plot show that learning something in 10 epochs alredy, is indeed possible; code is working as intended, performance on test set (measured **only** by per-class accuracy) is not acceptable: 0% for some classes. Per-class accuracy for class "BENIGN" was the highest among all (by far): this is an indication that there is a clear and recognizable distinction between DoS and BENIGN traffic.

Fallback plan

To increase performance in the MCCT I tried to create more complex models (in parameters number): that didn't help a lot.
At this point I started exploring BCT dataset and models to resolve the (possibly) easier task. I started from scratch again with a small MLP model similar to the one used in the first phase (512, 256, 128, 64, 2 neurons in fully connected layers), with Tanh as my only activation function. I also used an "intelligent" feature reduction technique called "ANOVA" (**Analysis of Variance**).
To simplify learning I have to remove those features that are independent from the target: how do I find the "dependence"? With ANOVA. ANOVA can be used when one variable is numeric and one is categorical, such as numerical input variables and a classification target variable in a classification task.

```
In [ ]: #NON-EXECUTABLE CODE SAMPLE
        from sklearn.feature_selection import SelectKBest, f_classif

X=df_copy.drop(['Label'], axis=1)
Y=df_copy['Label']

fvalue_Best = SelectKBest(f_classif, k=20)
X_kbest = fvalue_Best.fit_transform(X, Y)
mask=fvalue_Best.get_support()

print('Remaining columns:')
for m,c in zip(mask, cols):
    if m:
        print(c)

X=X_kbest
```

As provided by `sklearn.featture_selection` with option `f_classif`, `SelectKBest` performs one-way ANOVA F-value test, ignoring constant/categorical feature columns. Using the result of `f_classif`, keeps highest scoring k features (in my case 20).

$$F\_score = \frac{\text{variation between sample means (MSB)}}{\text{variation within the samples (MSW)}}$$

$$MSB = \frac{\text{Sum of squares between the group (SSB)}}{DFb}$$

$$\text{where}$$

$$SSB = \sum (X_i - X_t)^2 \text{ with } X_i \text{ mean of group } i \text{ and } X_t \text{ mean of all the observations}$$

$$DFb = \text{ degrees of freedom between } = \text{ total number of observations in all the groups} - 1$$

$$MSW = \frac{\text{Sum of squares within the group (SSW)}}{DFw}$$

where

$DFw =$ degrees of freedom within $= N - K$ with K number of groups and N total number of observations in all

$$SSW = \sum (X_{i,j} - X_j)^2 \text{ with } X_{i,j} \text{ is the observation of each group } j$$

⚠ **WARNING**

An observation is not a sample of the dataset: 1 dataset row with q features has q observations! So N is $\text{number of samples} * \text{number of features (groups)}$ and K is $\text{number of features}$

> *For details:*
> ***https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.Sele***
> ***https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.f_classif.html#skl***
> ***https://scikit-learn.org/stable/modules/feature_selection.html#univariate-featu***

Training on 20 features only, was extremely beneficial:

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

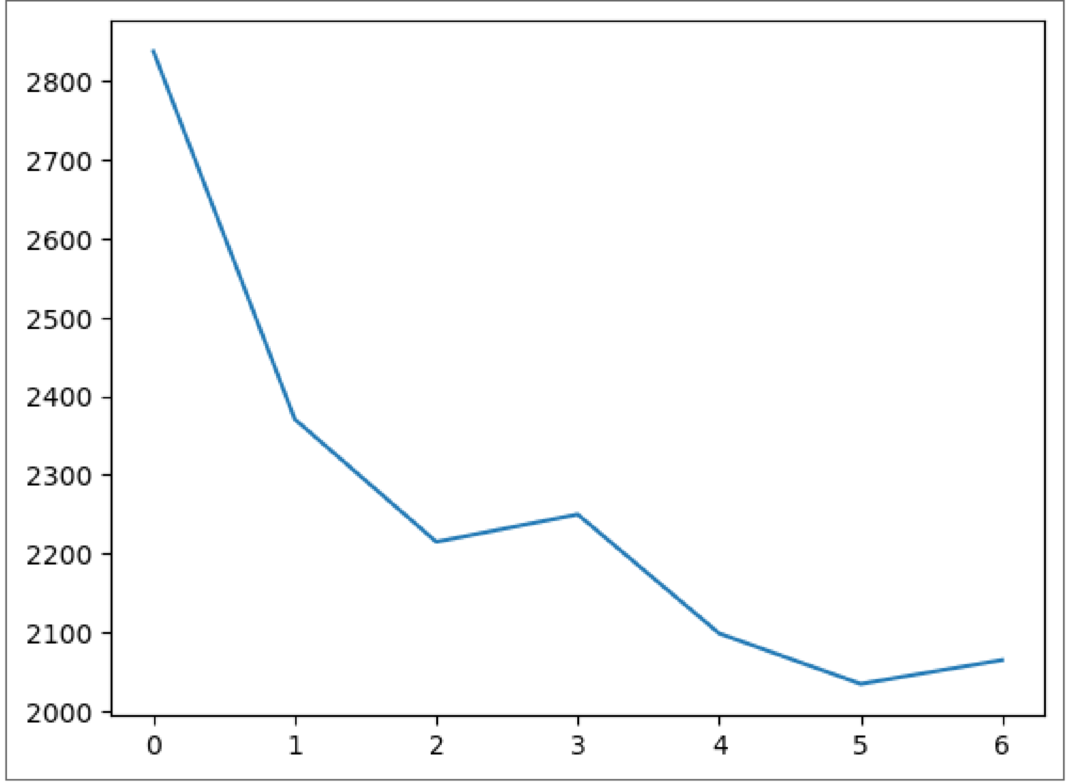
BUT MAYBE I CAN DO WITHOUT ANOVA... AUTOENCODERS?

```
In [ ]: #NON-EXECUTABLE CODE SAMPLE
        class ShieldNetAE(torch.nn.Module):
```

```
    def __init__(self):
        super(AttackNet, self).__init__()
        self.encoder=torch.nn.Sequential(
            torch.nn.Linear(70, 64),
            torch.nn.Tanh(),
            torch.nn.Linear(64, 32),
            torch.nn.Tanh(),
            torch.nn.Linear(32, 20) #as ANOVA k=20
            )

        self.decoder=torch.nn.Sequential(
            torch.nn.Linear(20, 32),
            torch.nn.Tanh(),
            torch.nn.Linear(32, 64),
            torch.nn.Tanh(),
            torch.nn.Linear(64, 70)
            )

        self.classifier=torch.nn.Sequential(
            torch.nn.Linear(20, 512),
            torch.nn.ReLU(),
            torch.nn.Linear(512, 256),
            torch.nn.ReLU(),
            torch.nn.Linear(256, 128),
            torch.nn.Tanh(),
            torch.nn.Linear(128, 64),
            torch.nn.Tanh(),
            torch.nn.Linear(64, 2),
            torch.nn.Sigmoid(),
            torch.nn.LogSoftmax(dim=1)
            )

    def forward(self, x):
        encoded=self.encoder(x)
        decoded=self.decoder(encoded)
        out=self.classifier(encoded)
        return decoded, out
```

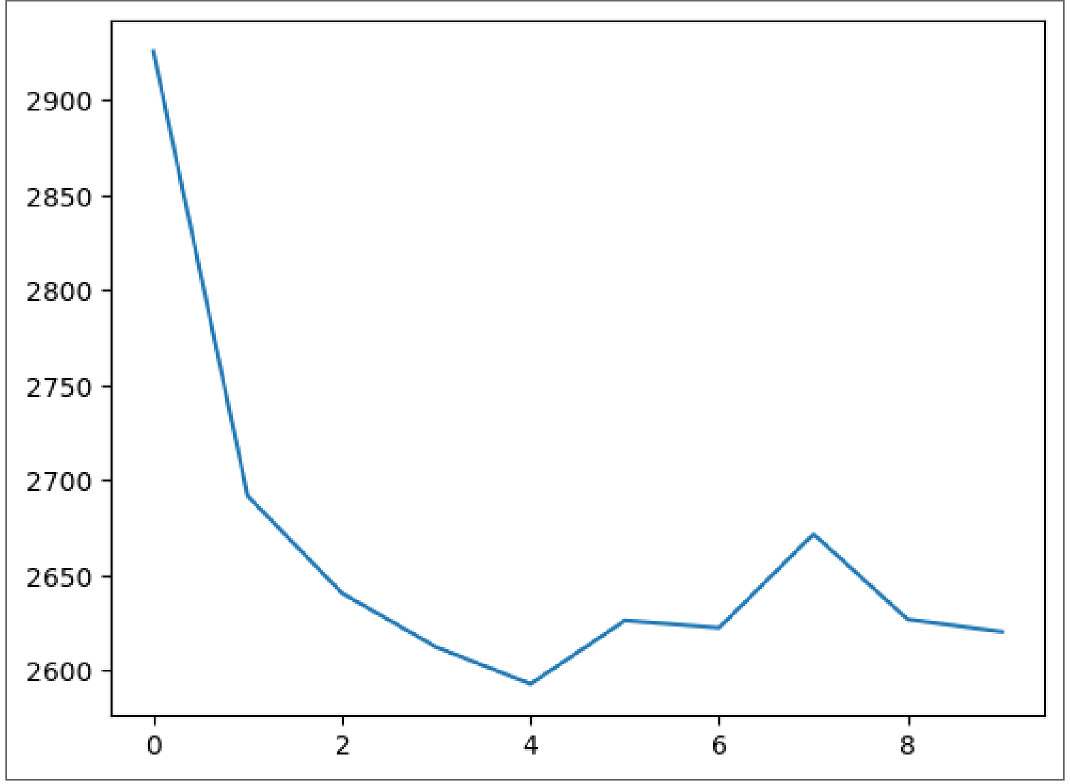
```
In [ ]: #NON-EXECUTABLE CODE SAMPLE
        ...

criterion1 = torch.nn.MSELoss().to(device)
criterion2 = torch.nn.CrossEntropyLoss().to(device)

...

for epoch in range(epochs):
    for x,y in loader:
        x=x.to(device)
        y=y.to(device)
        # reset optimizer
        opt.zero_grad()

        decoded, output = model(x)
        loss1 = criterion1(decoded.to(device), x)
        loss2 = criterion2(output.to(device), y)
        loss = loss1+loss2

        loss.backward()
        opt.step()
```

With an autoencoder we can simplify the BCT problem and obtain a similar performance to ANOVA in a similar number of epochs, but training **is definitely** slower: I have in fact to train encoder, decoder **and** classifier.
Loss plot below.



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

Same ANOVA approach on BCCT resulted in better-than-before, but very low performance, nontheless.
Loss plot below.


2900
2850
2800
2750
2700
2650
2600
0
2
4
6
8

Final models and results

In this section I present the final models so far, that archieved the best performances and could be taken as definitive.

BCT

After some more attemps I produced a final model by using:

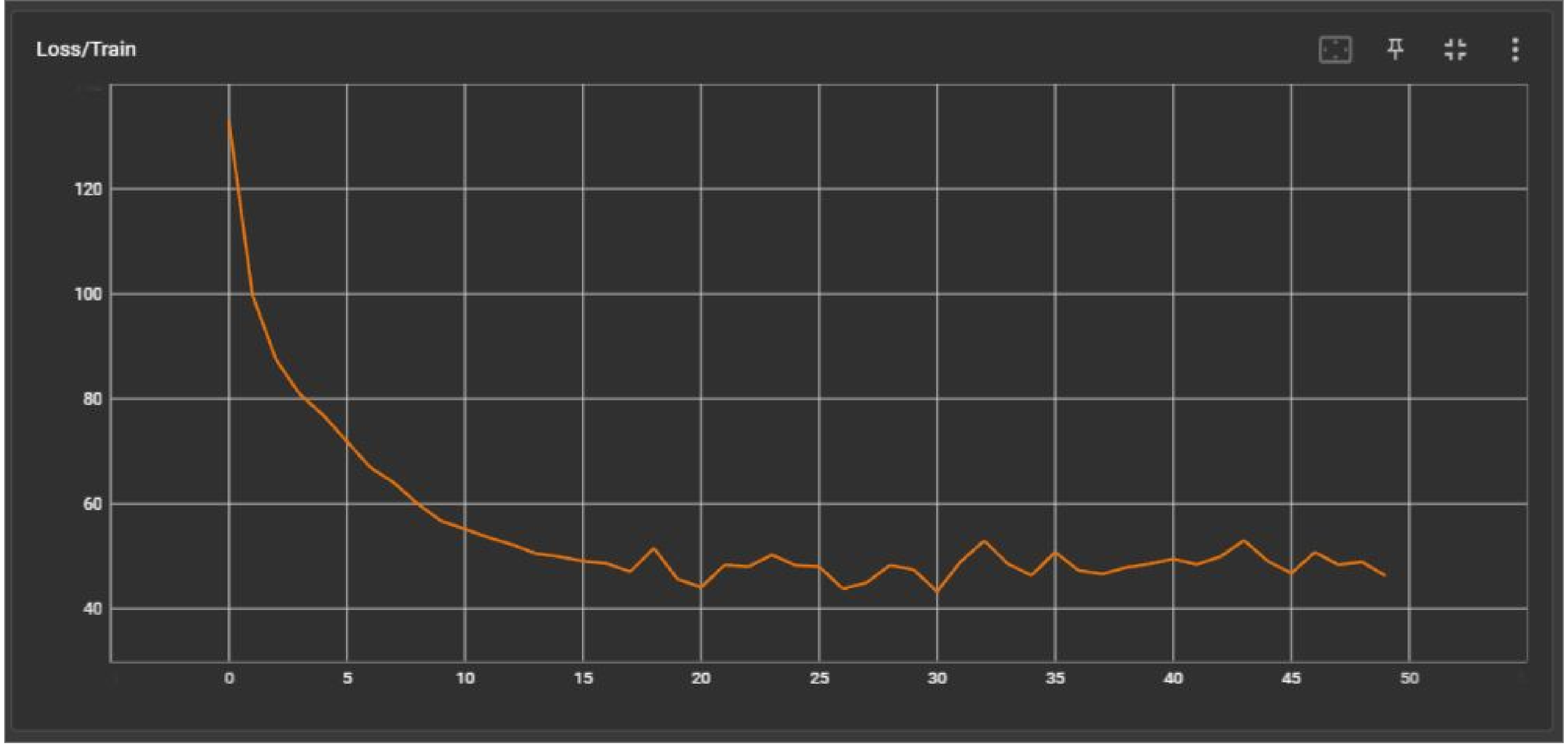
- ANOVA with k = 20;
- batch size = 100;
- `torch.nn.init.xavier_uniform_(module.weight)` weight initialization (best among default, `xavier_normal_`, `xavier_uniform_`, `kaiming_uniform_`, `kaiming_normal_`);
- structure reported below;
- 50 epochs;
- learning rate = 0.001;
- Adam optimizer with CrossEntropyLoss;
- learning rate optimizer `StepLR` with step_size = 10, gamma = 0.1;
- a rudimentary early stopping mechanism

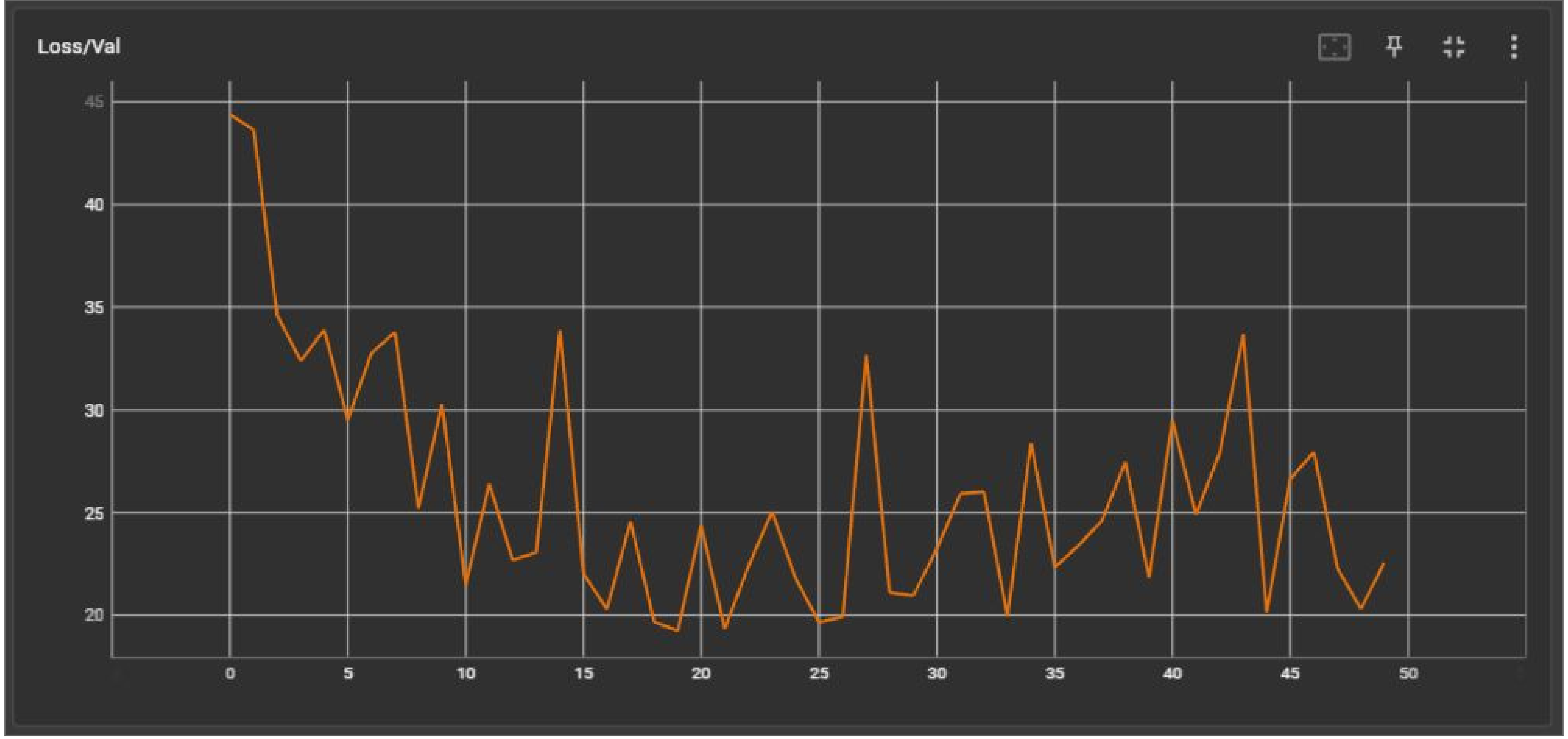
that performed really well.

```
In [ ]: #NON-EXECUTABLE CODE SAMPLE
        class AttackNetDefinitive(torch.nn.Module):

    def __init__(self):
        super(AttackNet, self).__init__()
        self.model=torch.nn.Sequential(
            torch.nn.Linear(20, 512),
            torch.nn.ReLU(),
            torch.nn.Linear(512, 256),
            torch.nn.ReLU(),
            torch.nn.Linear(256, 128),
            torch.nn.Tanh(),
            torch.nn.Linear(128, 64),
            torch.nn.Tanh(),
            torch.nn.Linear(64, 2),
            torch.nn.LogSoftmax(dim=1)
            )

    def forward(self, x):
        return self.model(x)
```

```
In [ ]: #NON-EXECUTABLE CODE SAMPLE
        def init_weights(module):
         if isinstance(module, torch.nn.Linear):
             torch.nn.init.xavier_uniform_(module.weight)
             if module.bias is not None:
                 module.bias.data.fill_(0.0001)


...

model = AttackNet()
model.apply(init_weights)

...
```

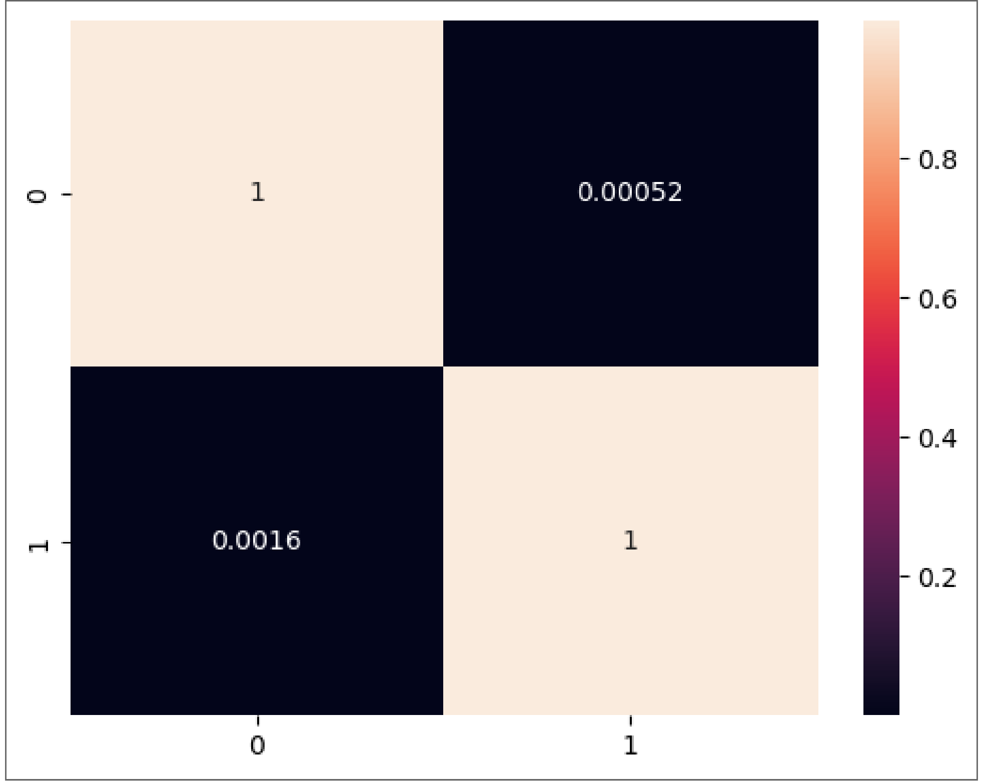
In particular, I managed to lower the loss significantly:


Loss/Train
120
100
80
60
40
0
5
10
15
20
25
30
35
40
45
50



And the performance was sky-high:

```
Precision:  tensor(0.9995)
Recall:  tensor(0.9984)
F1-score:  tensor(0.9989)
Overall Accuracy:  tensor(0.9989)
Per class accuracy:  [Benign 0.9994804904170463, DoS 0.998358450377156]
```

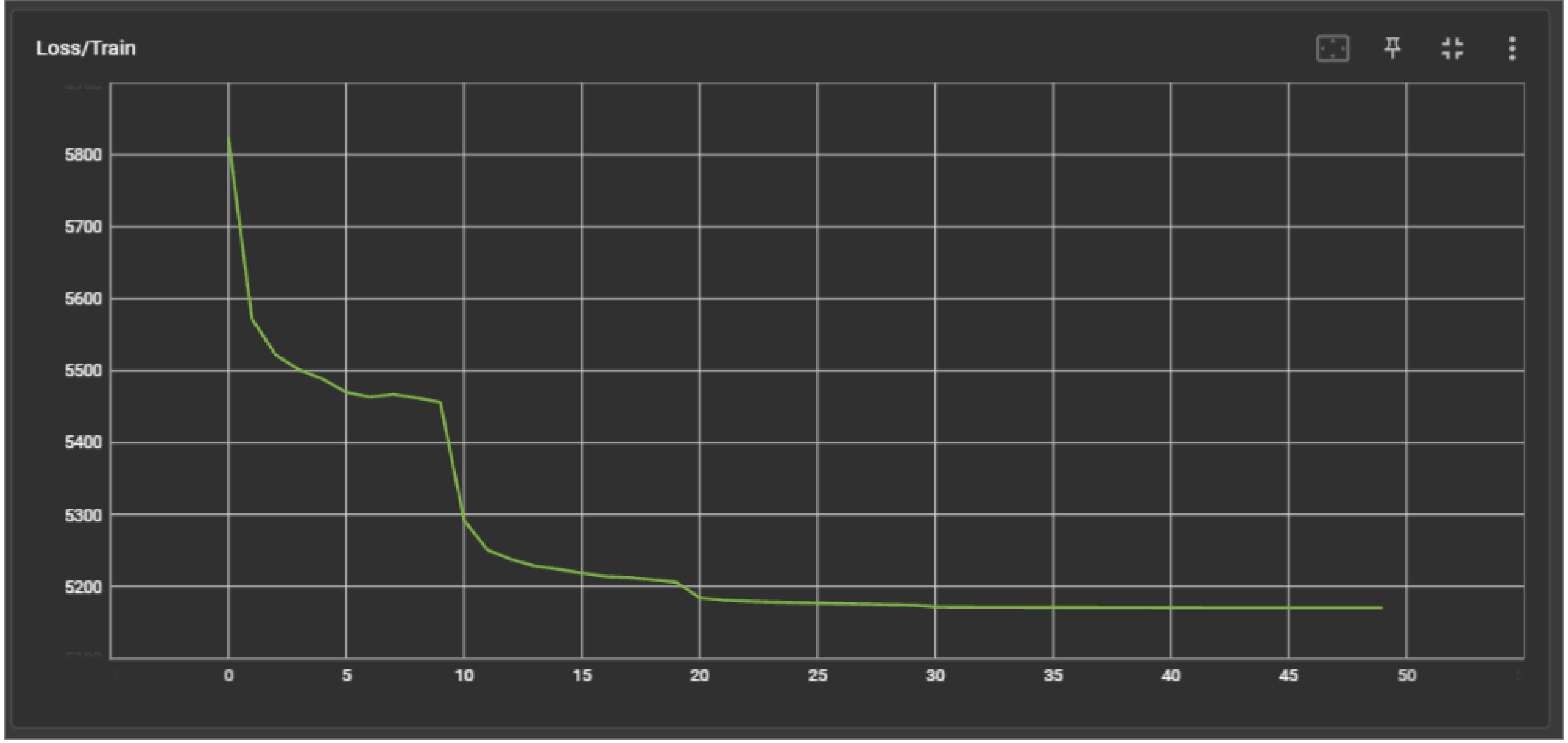

1
0.00052
0.0016
1
0
1
0
1
0.8
0.6
0.4
0.2

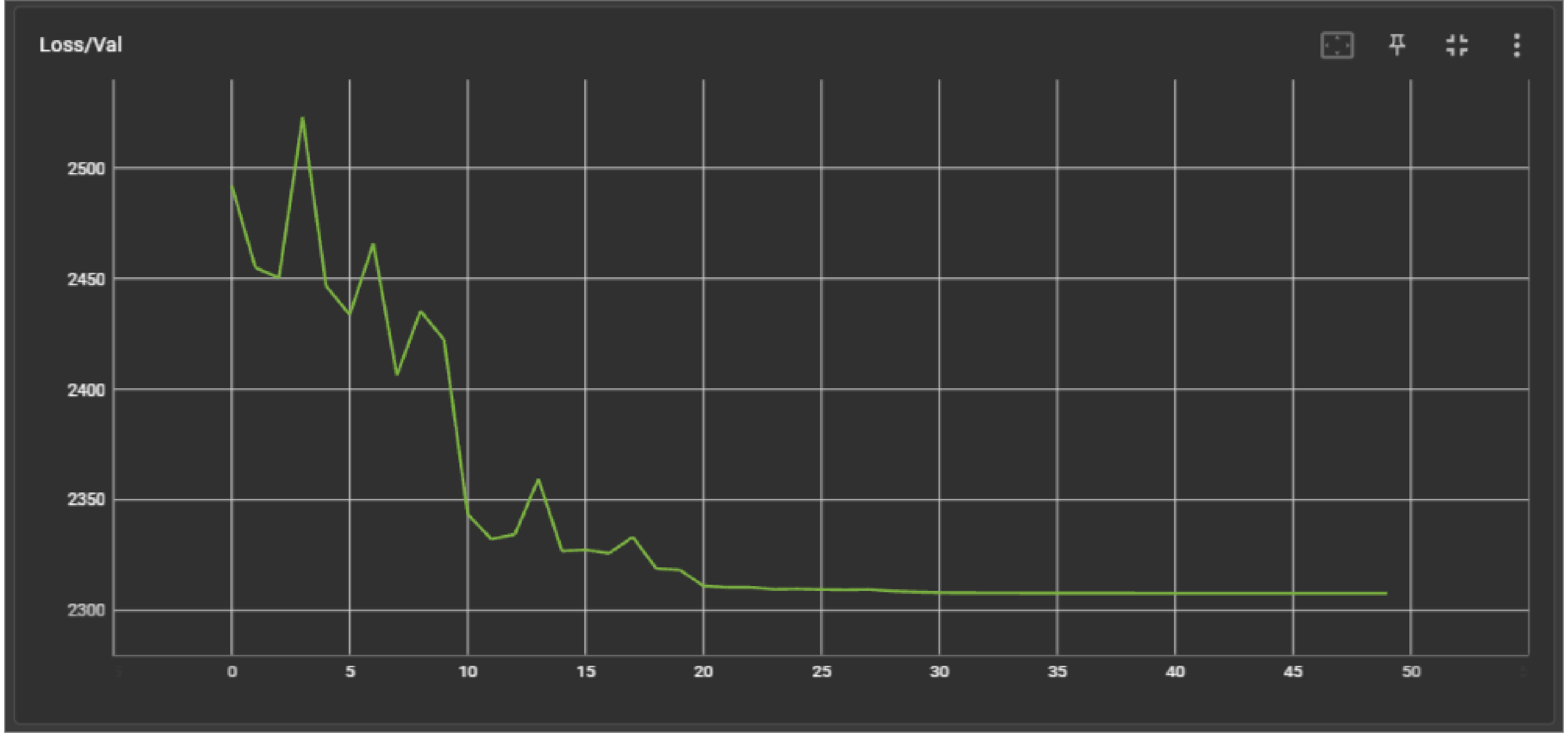
MCCT

The model and the hyperparameters for MCCT are similar to the ones presented for BCT, with some necessary adjustments to the morphology of the ANN. The only difference is that I decided to **not** use ANOVA.

```
In [ ]: #NON-EXECUTABLE CODE SAMPLE
        class AttackNet(torch.nn.Module):

    def __init__(self):
        super(AttackNet, self).__init__()
        self.model=torch.nn.Sequential(
            torch.nn.Linear(67, 512),
            torch.nn.ReLU(),
            torch.nn.Linear(512, 256),
            torch.nn.ReLU(),
            torch.nn.Linear(256, 128),
            torch.nn.Tanh(),
            torch.nn.Linear(128, 64),
            torch.nn.Tanh(),
            torch.nn.Linear(64, 12),
            torch.nn.LogSoftmax(dim=1)
            )

    def forward(self, x):
        return self.model(x)
```

Unlike BCT, MCCT has room for improvement. But, with the help of the scheduler, I managed to find a better local optimum, that is clearly shown in the plots:


Loss/Train
5800
5700
5600
5500
5400
5300
5200
0
5
10
15
20
25
30
35
40
45
50



Performance was considerably better than previous attempts, but some classes are very difficult to classify correctly:

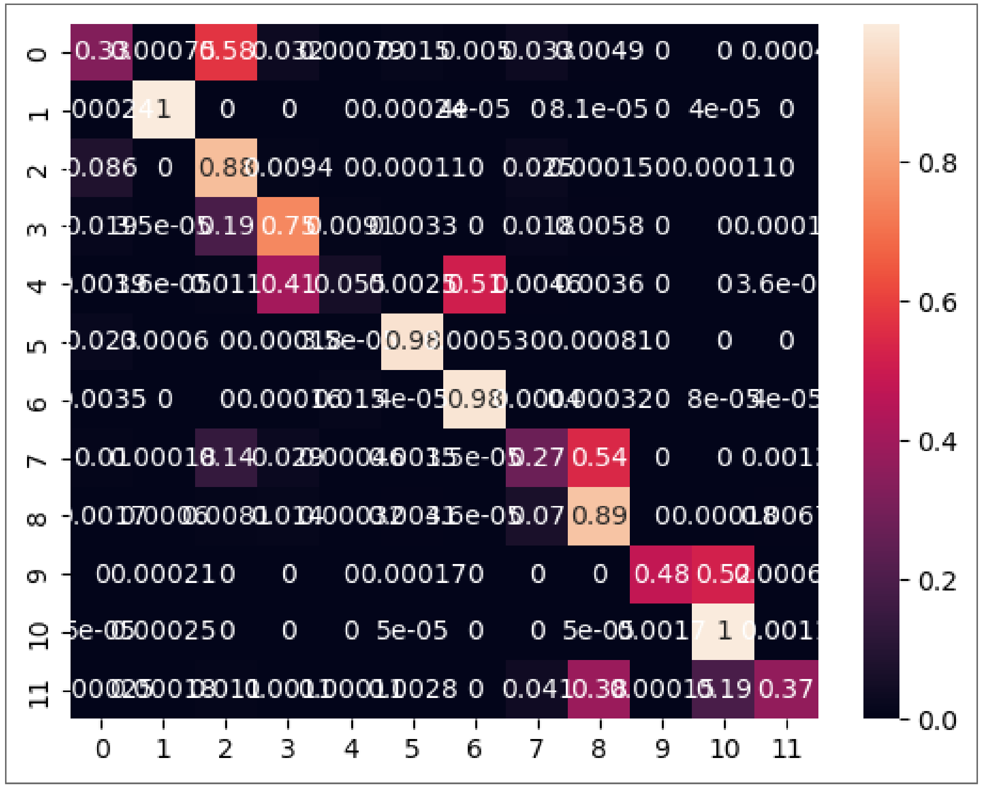
```
Precision:  tensor(0.7203)
Recall:  tensor(0.6648)
F1-score:  tensor(0.6208)
Overall Accuracy:  tensor(0.6648)
Per class accuracy:  [0.3305182203694387, Benign 0.999355150733516, 0.8797976792621244, 0.7496016712105654, 0.05497496189854126,
0.9750008839857148, 0.9800937237153042, 0.273768573353665,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

⚠ **WARNING**

Note that, as anticipated, the distinction between DoS and Benign traffic is quite easy to do. This is justified by the almost perfect accuracy for class Benign

WARNING

(99.93%)!



Thank you for the attention