Projet ouverture

Authors
SAGDULLIN Damir
LAIOLO Léo
TABBAH Nicola

Introduction

This project aim to recognize attacks on connected vehicles using Deep Learning (DL).

Imports

Show code

Execution mode

This jupyter notebook can either be run on Google colab or on a local machine.

Please select your computation mode:

computation_mode: Google Colab

If you chose `Local` computation mode, please specify the path to the dataset folder.

path: " ./ "

Show code

```
Connecting to Google Drive...
Mounted at /content/drive
/content/drive/Shareddrives/ING3 Ouverture/Project
```

Dataset

Our dataset is quite big. Generating it takes a lot of time. This is why we saved it in a CSV file.

To discribe the treatments applied to the original dataset, we extract only a piece of the original dataset and pre-treat it. In this section, you can follow data transformation on a little portion of the dataset. At the end, we download the full pretreated dataset to train our models.

Our dataset is a part of the Vehicular Reference Misbehavior (VeReMi), built specifically for testing V2X security. We uses the received message historics from 2260 connected vehicles.

Some of them send malicious messages of 19 different types.

Connected vehicles exchange lots of messages, to localize themselves or get traffic state, with other vehicles, roadside elements and infrastructures. In this study, we only work on exchanges between vehicles. This kind of message is declared with a `type:3`. So we first select all the `type:3` messages from our vehicles.

Import data

Our dataset is made of two parts:

1. The labels
2. The messages per vehicles

Labels are a list of message sender vehicle labelised as `0` for benign and `X` for malicious, where `X` corresponds to the index of the attack. We import this list of vehicles from a CSV file.

The second part of the dataset is composed of 2260 vehicles' message historics. We import them and keep only the `type:3` messages, exchanged by vehicles. Here is an example of a message content:


```
{
  "type":3,
  "rcvTime":50427.66028717679,
  "sendTime":50427.66028717679,
  "sender":57,
  "senderPseudo":10575,
  "messageID":67985,
  "pos":[983.2263964536893,910.8786100947851,0.0],
  "pos_noise":[3.6170773833523657,3.9726270832479986,0.0],
  "spd":[-8.160545717352005,-5.838995406773277,0.0],
  "spd_noise":[-0.013202827672876944,-0.00944703121125735,0.0],
  "acl":[-0.1056377948217638,-0.07553041612577578,0.0],
  "acl_noise":[0.001235094686840946,0.0021069253674089584,0.0],
  "hed":[-0.8479991671178503,-0.5299975590202584,0.0],
  "hed_noise":[24.07432760428006,19.08668548660413,0.0]
}
```


Show code

Choose Dataset

To see an example of pretreatments, you can select a directory containing json message historics and the number of files to load.

dataset_directory: VeReMi_64800_68400_2022-9-11_12_51_1/

number_vehicles: 5

Show code

```
Importing Jsons...
    File 1/5: traceJSON-110589-110587-A0-66720-18.json
     | - > Concatenation...
     \ - > Done
    File 2/5: traceJSON-110553-110551-A16-66714-18.json
     | - > Concatenation...
     \ - > Done
    File 3/5: traceJSON-110661-110659-A0-66733-18.json
     | - > Concatenation...
     \ - > Done
    File 4/5: traceJSON-110715-110713-A5-66742-18.json
     | - > Concatenation...
     \ - > Done
    File 5/5: traceJSON-110667-110665-A0-66733-18.json
     | - > Concatenation...
     \ - > Done
STOP IMPORTATION: Json limit reached.
```

	file	receiver	omnet_module_id	attacker_type	
1	traceJSON-110589-110587-A0-66720-18.json	110589	110587	A0	66720
2	traceJSON-110589-110587-A0-66720-18.json	110589	110587	A0	66720
3	traceJSON-110589-110587-A0-66720-18.json	110589	110587	A0	66720
4	traceJSON-110589-110587-A0-66720-18.json	110589	110587	A0	66720
5	traceJSON-110589-110587-A0-66720-18.json	110589	110587	A0	66720

Show code

```
<class 'pandas.core.frame.DataFrame'>
Index: 5880 entries, 1 to 1183
Data columns (total 17 columns):
 #   Column             Non-Null Count  Dtype
```

```
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   file             5880 non-null   object
 1   receiver         5880 non-null   object
 2   omnet_module_id  5880 non-null   object
 3   attacker_type    5880 non-null   object
 4   rcvTime          5880 non-null   float64
 5   sendTime         5880 non-null   float64
 6   sender           5880 non-null   float64
 7   senderPseudo     5880 non-null   float64
 8   messageID        5880 non-null   float64
 9   pos              5880 non-null   object
 10  pos_noise        5880 non-null   object
 11  spd              5880 non-null   object
 12  spd_noise        5880 non-null   object
 13  acl              5880 non-null   object
 14  acl_noise        5880 non-null   object
 15  hed              5880 non-null   object
 16  hed_noise        5880 non-null   object
dtypes: float64(5), object(12)
memory usage: 955.9+ KB
```

Generate sender label

The labels need to be generated from the json files. To do so, we look for the receiver vehicles categorized as malicious, we extract their type of attack then we save those data into a special dataset.

Show code

```
<class 'pandas.core.frame.DataFrame'>
Index: 5 entries, 1 to 1
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   sender  5 non-null      int64
 1   label   5 non-null      int64
dtypes: int64(2)
memory usage: 120.0 bytes
```

Merge BSM messages and senders labels data

Once our data imported, we centralize all the data to easily pretreat them. To do so, we merge the 2 parts of our dataset.

Because we are working on a sample of the dataset, some information might be missing. To ensure that the rest of the treatments will run properly, we delete those incomplete rows.

Show code

file	receiver	omnet_module_id	attacker_type

0	traceJSON-110589-110587-A0-66720-18.json	110589	110587	A0	66720
1	traceJSON-110589-110587-A0-66720-18.json	110589	110587	A0	66720
2	traceJSON-110589-110587-A0-66720-18.json	110589	110587	A0	66720
3	traceJSON-110589-110587-A0-66720-18.json	110589	110587	A0	66720
4	traceJSON-110589-110587-A0-66720-18.json	110589	110587	A0	66720

Next steps: View recommended plots

Split list features

We see in the tables displayed in the above section that some message data are lists of coordinates, like the position `pos`. Our model needs flatten data, not nested lists. Therefore, we split those coordinates into columns `x`, `y` and `z`, spliting `pos` into `pos_x`, `pos_y` and `pos_z`.

Show code

	file	receiver	omnet_module_id	attacker_type	
0	traceJSON-110589-110587-A0-66720-18.json	110589	110587	A0	66720
1	traceJSON-110589-110587-A0-66720-18.json	110589	110587	A0	66720
2	traceJSON-110589-110587-A0-66720-18.json	110589	110587	A0	66720
3	traceJSON-110589-110587-A0-66720-18.json	110589	110587	A0	66720
4	traceJSON-110589-110587-A0-66720-18.json	110589	110587	A0	66720

5 rows × 34 columns

Show code

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 257 entries, 0 to 256
Data columns (total 34 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   file             257 non-null    object
 1   receiver         257 non-null    object
 2   omnet_module_id  257 non-null    object
 3   attacker_type    257 non-null    object
 4   rcvTime          257 non-null    float64
 5   sendTime         257 non-null    float64
 6   sender           257 non-null    float64
 7   senderPseudo     257 non-null    float64
 8   messageID        257 non-null    float64
 9   label            257 non-null    int64
 10  pos_x            257 non-null    float64
 11  pos_y            257 non-null    float64
 12  pos_z            257 non-null    float64
 13  pos_noise_x      257 non-null    float64
 14  pos_noise_y      257 non-null    float64
 15  pos_noise_z      257 non-null    float64
 16  spd_x            257 non-null    float64
 17  spd_y            257 non-null    float64
 18  spd_z            257 non-null    float64
 19  spd_noise_x      257 non-null    float64
 20  spd_noise_y      257 non-null    float64
 21  spd_noise_z      257 non-null    float64
 22  acl_x            257 non-null    float64
 23  acl_y            257 non-null    float64
 24  acl_z            257 non-null    float64
 25  acl_noise_x      257 non-null    float64
 26  acl_noise_y      257 non-null    float64
 27  acl_noise_z      257 non-null    float64
 28  hed_x            257 non-null    float64
 29  hed_y            257 non-null    float64
 30  hed_z            257 non-null    float64
 31  hed_noise_x      257 non-null    float64
 32  hed_noise_y      257 non-null    float64
 33  hed_noise_z      257 non-null    float64
dtypes: float64(29), int64(1), object(4)
memory usage: 68.4+ KB
```

Drop irrelevant cols

Looking at the message data, we see that some information are irrelevant. Indeed, we are about to split our dataset on receiver vehicles. They can be identified by each one of the following data:

- `sender`, the message sender is the vehicle;
- `file`, each historic file belongs to one and only vehicle;
- `omnet_module_id`, the vehicles' id on the network didn't change during the message recording.

Once our dataset split, those columns would be filled with the same value, making it irrelevant for the model.

There are also information that couldn't be interpreted by the model and wouldn't help it to understand the messages. Those are listed below:

- `senderPseudo`, the pseudo of the vehicles are integers aiming for sender differenciation;
- `messageID`, id serve to distinguish messages that could contain the same information.

Those information are generated by the sender's communication protocols. They can be faked but are to hard to recognize, even for human. Keeping those data would brings more difficulties to understand the message than dropping it.

Finaly, the `attacker_type` indicates if the receiver vehicle is malicious (`A13`) or not (`A0`). It is a data related to the receiver vehicle so it is useless to understand received message in our case.

Show code

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 257 entries, 0 to 256
Data columns (total 28 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   receiver     257 non-null    object
 1   rcvTime      257 non-null    float64
 2   sendTime     257 non-null    float64
 3   label        257 non-null    int64
 4   pos_x        257 non-null    float64
 5   pos_y        257 non-null    float64
 6   pos_z        257 non-null    float64
 7   pos_noise_x  257 non-null    float64
 8   pos_noise_y  257 non-null    float64
 9   pos_noise_z  257 non-null    float64
 10  spd_x        257 non-null    float64
 11  spd_y        257 non-null    float64
 12  spd_z        257 non-null    float64
 13  spd_noise_x  257 non-null    float64
 14  spd_noise_y  257 non-null    float64
 15  spd_noise_z  257 non-null    float64
 16  acl_x        257 non-null    float64
 17  acl_y        257 non-null    float64
 18  acl_z        257 non-null    float64
 19  acl_noise_x  257 non-null    float64
 20  acl_noise_y  257 non-null    float64
 21  acl_noise_z  257 non-null    float64
 22  hed_x        257 non-null    float64
 23  hed_y        257 non-null    float64
 24  hed_z        257 non-null    float64
 25  hed_noise_x  257 non-null    float64
 26  hed_noise_y  257 non-null    float64
 27  hed_noise_z  257 non-null    float64
```

```
dtypes: float64(26), int64(1), object(1)
memory usage: 56.3+ KB
```

Scale data

To improve our model efficiency, we scale the data, excepted the labels.

Show code

	rcvTime	sendTime	label	pos_x	pos_y	pos_z	po
count	257.000000	257.000000	257.000000	257.000000	257.000000	257.0	2
mean	0.416162	0.416162	5.579767	0.340015	0.483747	0.0	
std	0.362934	0.362934	7.382574	0.328630	0.346629	0.0	
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.0	
25%	0.070633	0.070633	0.000000	0.059629	0.138644	0.0	
50%	0.250258	0.250258	0.000000	0.085371	0.375982	0.0	
75%	0.766824	0.766824	16.000000	0.650239	0.822959	0.0	
max	1.000000	1.000000	16.000000	1.000000	1.000000	0.0	

8 rows × 27 columns

Load full dataset

In the previous section, we saw all the treatments applied to a sample of the original dataset. Now, we load the complete pretreated dataset. This last one will be used to train the proposed model.

Show code

	receiver	rcvTime	sendTime	label	pos
count	2.299626e+06	2.299626e+06	2.299626e+06	2.299626e+06	2.299626e+
mean	1.097735e+05	4.396304e-01	4.396304e-01	3.441680e+00	3.409545e-
std	3.884251e+03	3.143329e-01	3.143329e-01	5.375860e+00	2.619478e-
min	1.031670e+05	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+
25%	1.064790e+05	1.581104e-01	1.581104e-01	0.000000e+00	1.344144e-
50%	1.094850e+05	3.057054e-01	3.057054e-01	0.000000e+00	1.841866e-
75%	1.120610e+05	7.499819e-01	7.499819e-01	6.000000e+00	5.767359e-

75%	1.130610e+05	7.499819e-01	7.499819e-01	6.000000e+00	5.767359e-
max	1.167210e+05	1.000000e+00	1.000000e+00	1.900000e+01	1.000000e+

8 rows × 28 columns

Show code

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2299626 entries, 0 to 2299625
Data columns (total 28 columns):
 #   Column        Dtype  
---  ------        -----  
 0   receiver      int64  
 1   rcvTime       float64
 2   sendTime      float64
 3   label         int64  
 4   pos_x         float64
 5   pos_y         float64
 6   pos_z         float64
 7   pos_noise_x   float64
 8   pos_noise_y   float64
 9   pos_noise_z   float64
 10  spd_x         float64
 11  spd_y         float64
 12  spd_z         float64
 13  spd_noise_x   float64
 14  spd_noise_y   float64
 15  spd_noise_z   float64
 16  acl_x         float64
 17  acl_y         float64
 18  acl_z         float64
 19  acl_noise_x   float64
 20  acl_noise_y   float64
 21  acl_noise_z   float64
 22  hed_x         float64
 23  hed_y         float64
 24  hed_z         float64
 25  hed_noise_x   float64
 26  hed_noise_y   float64
 27  hed_noise_z   float64
dtypes: float64(26), int64(2)
memory usage: 491.3 MB
```

Show code

```
Attack classes : [ 0 17 15  6  8  9 19 11 14 16  7  1 10  3 12 18  2  5 13
```

DNN

In this project, we choose to implement a classic Deep Neural Network (DNN). Indeed, we observed in the TP4 that this model was more efficient than Recurrent Neural Network.

However, we didn't use Federated learning to compare this method with centralized training.

Generate train, validation and test

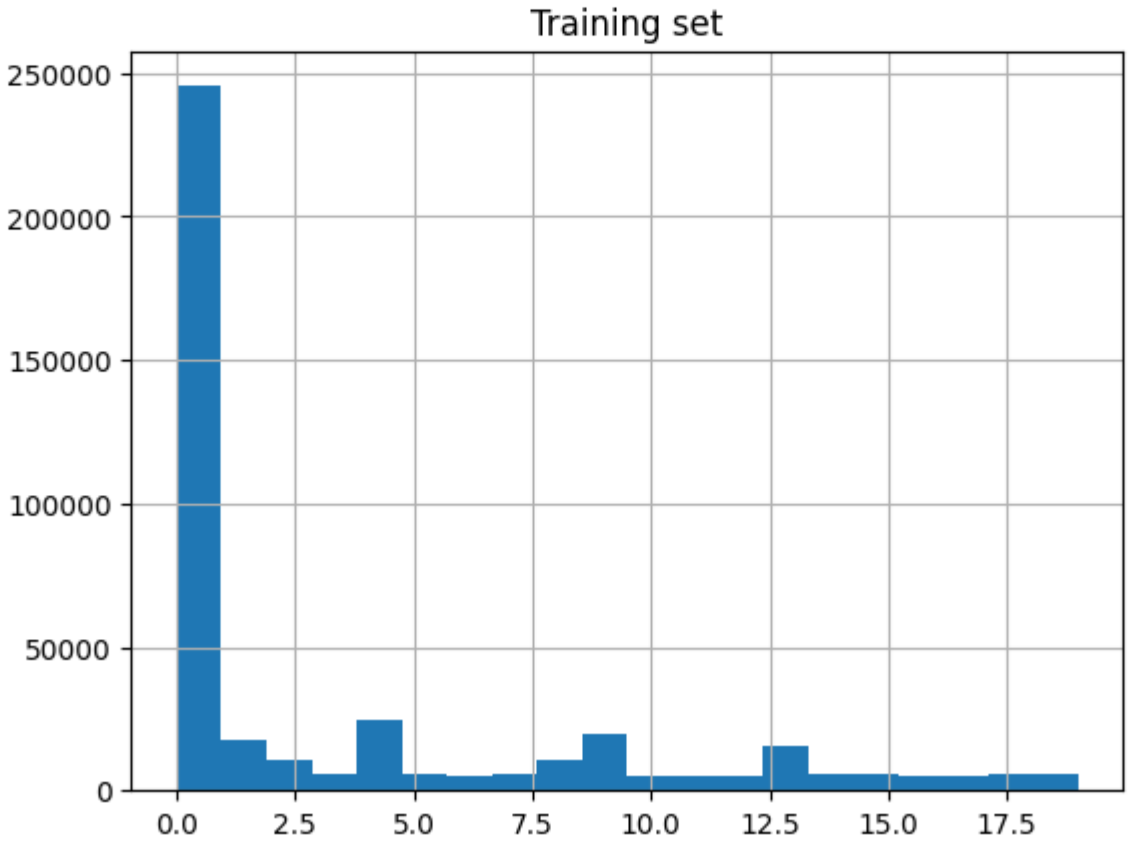
Though we pretreated the dataset, it still needs to be prepared for our model. We choose to train our Deep Learning model using the *Train, Validate and Test* method. Therefore, we need to generate subsets.

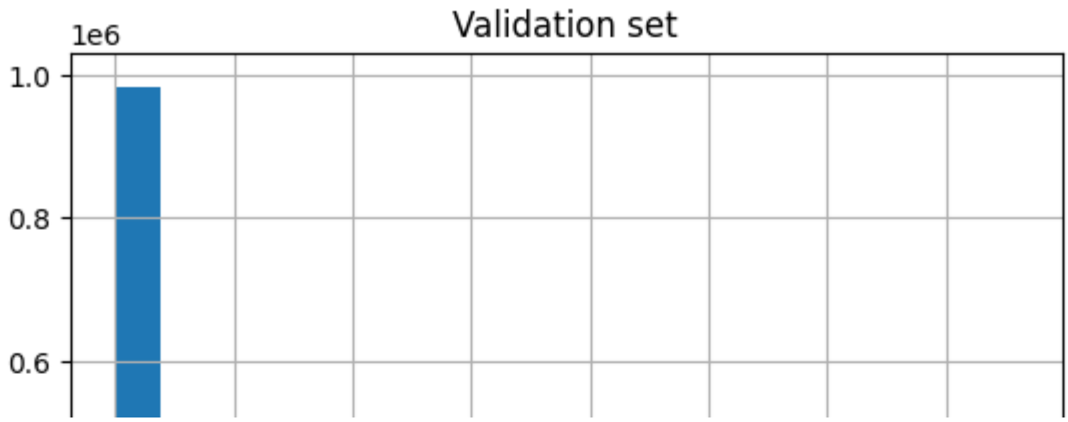
The datasets follow the distribution:

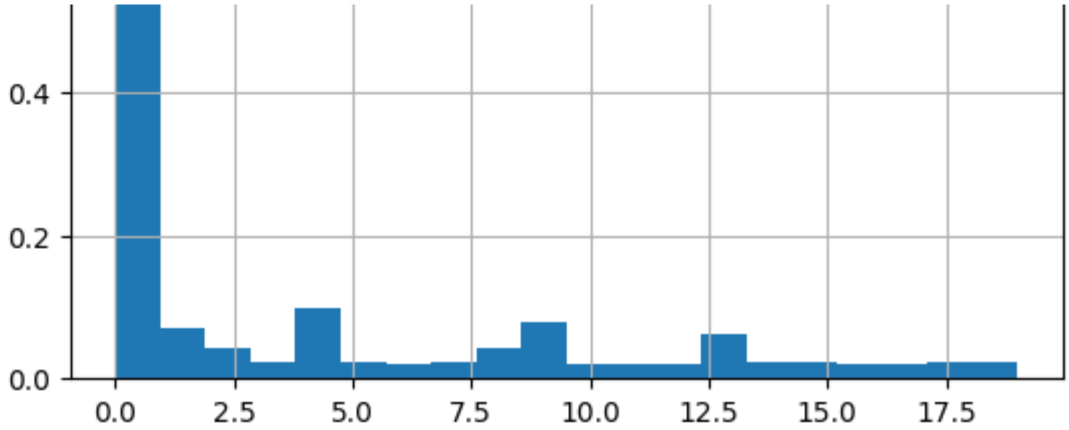
- 72% for training
- 18% for validation
- 10% for test

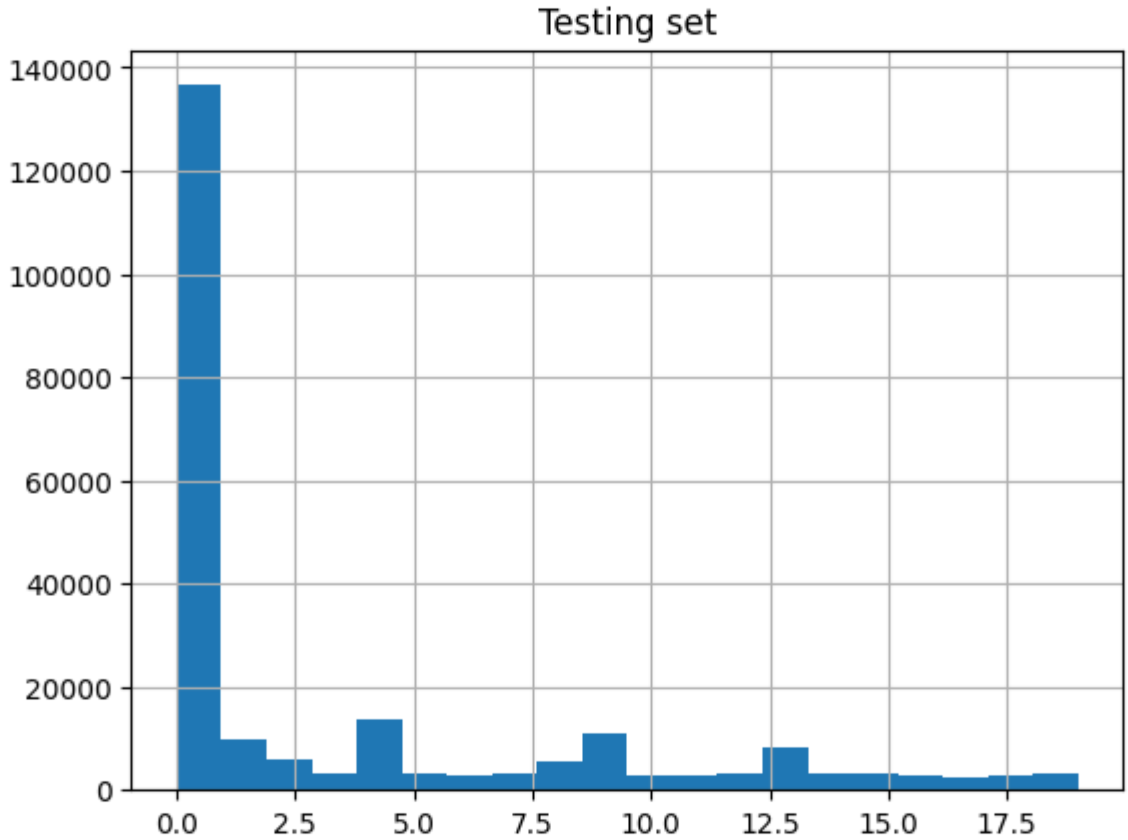
Show code

```
Labels distribution
```









Balancing dataset

Our datasets are almost ready. We now want to ensure that benign and malicious messages are balanced.

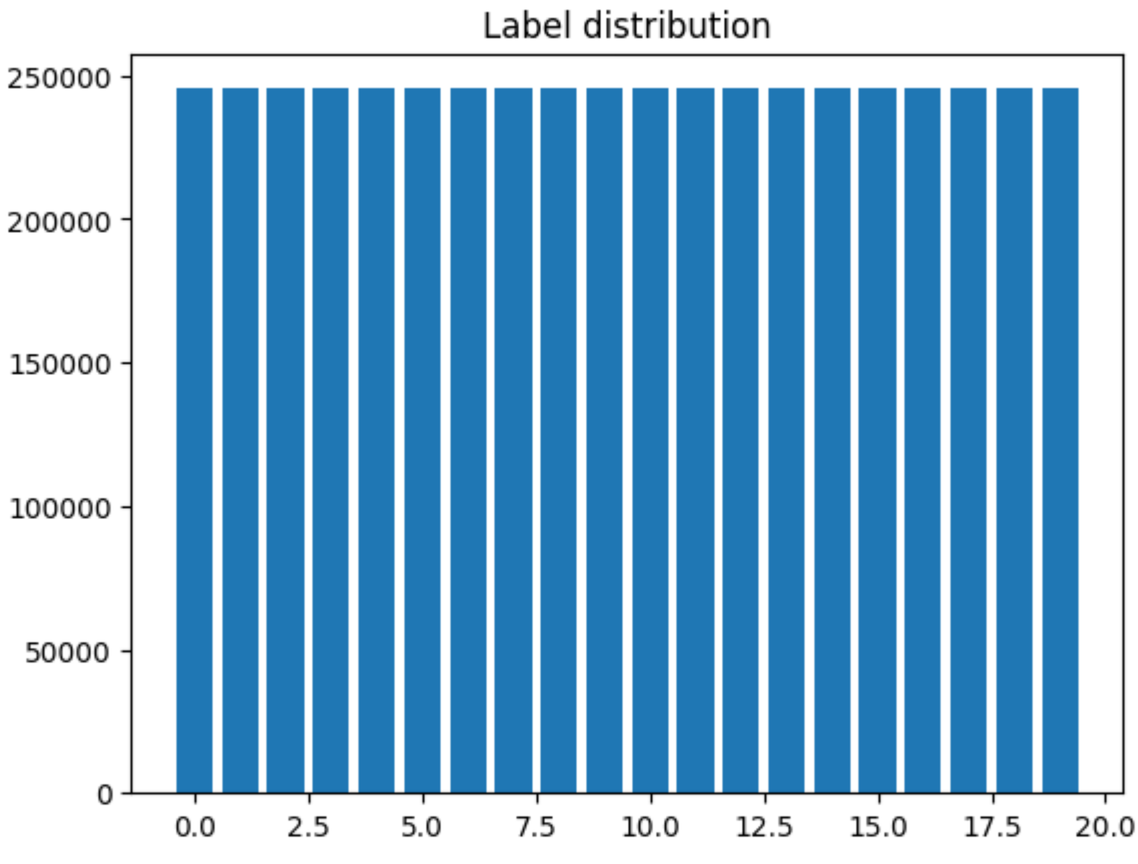
As we can see in the histograms above, none of the datasets are balanced. There is more normal messages than attacks. We need them to be more equitably distributed to optimize our model's performance.

One way to balance a dataset is to do *Oversampling*. This method consist to duplicates data from the minority class until all the classes have the same amount of elements.

Here, we used *Synthetic Minority Oversampling Technique* (SMOTE), which consists to generate new data with couples of close data. In it based on the K-NN technique.

Show code

```
Class=0, n=245370 (5.000%)
Class=13, n=245370 (5.000%)
Class=9, n=245370 (5.000%)
Class=4, n=245370 (5.000%)
Class=18, n=245370 (5.000%)
Class=6, n=245370 (5.000%)
Class=15, n=245370 (5.000%)
Class=3, n=245370 (5.000%)
Class=14, n=245370 (5.000%)
Class=1, n=245370 (5.000%)
Class=7, n=245370 (5.000%)
Class=5, n=245370 (5.000%)
Class=2, n=245370 (5.000%)
Class=8, n=245370 (5.000%)
Class=17, n=245370 (5.000%)
Class=19, n=245370 (5.000%)
Class=16, n=245370 (5.000%)
Class=11, n=245370 (5.000%)
Class=10, n=245370 (5.000%)
Class=12, n=245370 (5.000%)
```



Generating model

We implemented Deep Neural Network (`DNN`). A DNN is a Neural Network that has more than one hidden layer. Here, it has the following architecture:

Layer	Number of neurones	Activation
Input	N	ReLU
Feed forward 1	128	ReLU

Feed forward 2	64	ReLU
Feed forward 3	32	ReLU
Output	m	Softmax

where N is the number of features fed to the network and m is the number of classes. In this project, m is equal to 20 .

This is the same architecture used in for Federated Learning, excepted the last layer.

Show code

Load pretrained weights

We trained the same model on different numbers of epochs. You can choose them by writing a file name from the folder `Weights/`. This way, you can see the model's evolution.

Input the file name contening saved weights:

load_weight_file: " dnn_50epoch.keras "

Show code

Training

To train our model, we define two hyper parameters:

- `epochs` , the number of training epochs for the local models
- `pow2_batch_size` , power applied to 2 to define the number of messages used for in a batch

We also used an early stopping method so the local models doesn't do overfitting.

Be aware that because of the huge dataset, **models take a long time to train**. We advise you to save it in a file after each training.

Set training parameters:

epochs: 1

pow2_batch_size: 10

Show code

```
Batch size set to 1024 .
WARNING:tensorflow:5 out of the last 5 calls to <function _BaseOptimizer._
WARNING:tensorflow:6 out of the last 6 calls to <function _BaseOptimizer._
```

```
4793/4793 - 194s - loss: 0.7460 - accuracy: 0.7558 - val_loss: 1.5598 - va
<keras.src.callbacks.History at 0x7c2e8807ee30>
```

Saving

Input the file name where the model's weights will be saved:

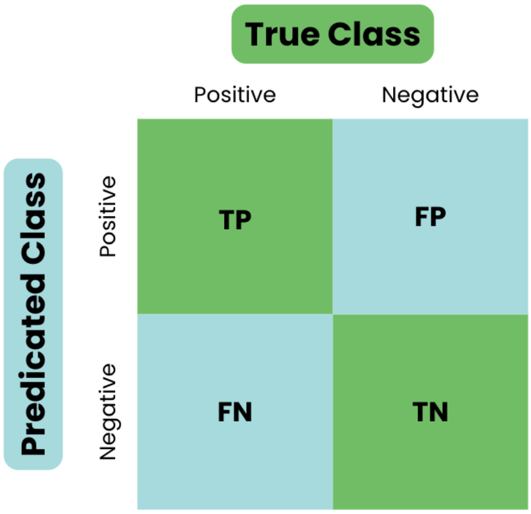
save_weight_file: " Insert text here "

Show code

Evaluating

Now that our model has been trained, we can evaluate its performance on the testing set. We used the following classification metrics for evaluation:

- *Confusion matrix*, described with this table.



- *Precision*, described by the formula:

$$precision = \frac{TP}{TP + FP}$$

- *Recall*, described by the formula:

$$recall = \frac{TP}{TP + FN}$$

- *Accuracy*, described by the formula:

$$accuracy = \frac{TP + TN}{TP + FP + TN + FN}$$

- *F1-score*, described by the formula:

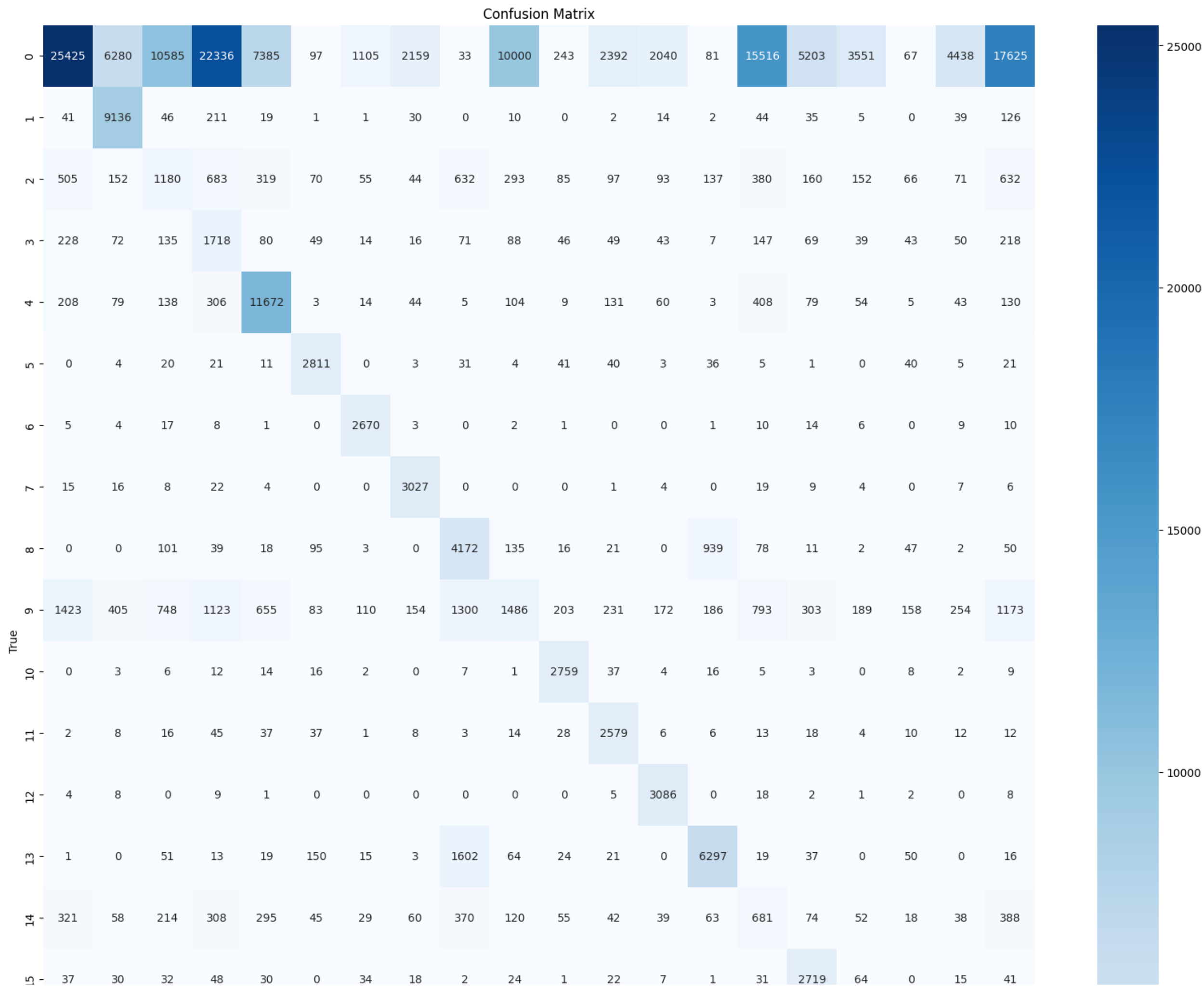
$$F1Score = 2 \times \frac{precision \times recall}{precision + recall}$$

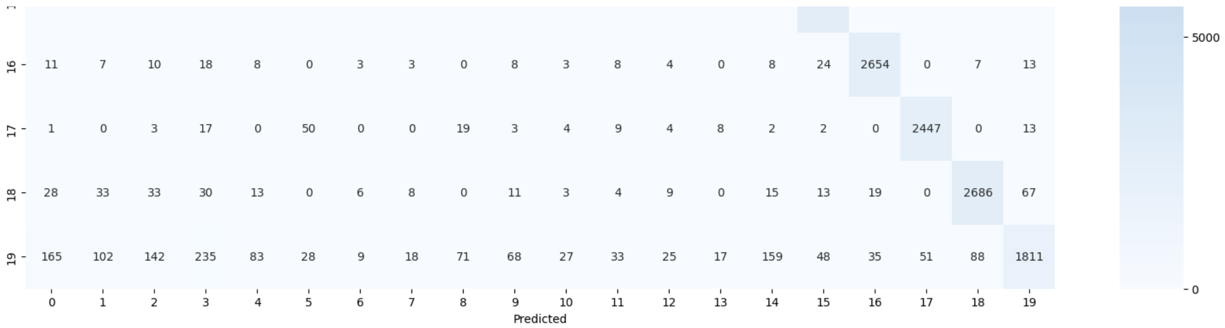
Show code

```
7187/7187 [==============================] - 35s 5ms/step
              precision    recall  f1-score   support

           0       0.89      0.19      0.31    136561
           1       0.56      0.94      0.70      9762
           2       0.09      0.20      0.12      5806
           3       0.06      0.54      0.11      3182
           4       0.56      0.86      0.68     13495
           5       0.80      0.91      0.85      3097
           6       0.66      0.97      0.78      2761
           7       0.54      0.96      0.69      3142
           8       0.50      0.73      0.59      5729
           9       0.12      0.13      0.13     11149
          10       0.78      0.95      0.86      2904
          11       0.45      0.90      0.60      2859
          12       0.55      0.98      0.70      3144
          13       0.81      0.75      0.78      8382
          14       0.04      0.21      0.06      3270
          15       0.31      0.86      0.45      3156
          16       0.39      0.95      0.55      2789
          17       0.81      0.95      0.87      2582
          18       0.35      0.90      0.50      2978
          19       0.08      0.56      0.14      3215

    accuracy                           0.40    229963
   macro avg       0.47      0.72      0.52    229963
weighted avg       0.71      0.40      0.40    229963
```




16 11 7 10 18 8 0 3 3 0 8 3 8 4 0 8 24 2654 0 7 13
17 1 0 3 17 0 50 0 0 19 3 4 9 4 8 2 2 0 2447 0 13
18 28 33 33 30 13 0 6 8 0 11 3 4 9 0 15 13 19 0 2686 67
19 165 102 142 235 83 28 9 18 71 68 27 33 25 17 159 48 35 51 88 1811
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Predicted
5000
0

Analysis

In this section, we discuss of the performance of our model and compare it to Federated Learning (FL).

The proposed DNN achieved an accuracy score of 40 %, on the testing set. Its confusion matrice also looks partly diagonal, meaning that the model didn't confused a lot of labels, except `0` . This label stands for the benign messages. It shows that attacks are quiet difficult to recognize, even for an AI trained on more than 250 000 messages.

The proposed DNN didn't achieve 50 % of accurency on the testing dataset, though the dataset was balanced and large enough. The principal reason of this low efficiency might be the number of epochs. Indeed, we only trained our model on 50 epochs because each one of them took a lot of time. We also could have made a more complex DNN but we couldn't compare it to those we used with FL.

Federated Learning reached better results in the last project, though training were longer. Indeed, FL needed 20 epochs per DNN and a local models fusion to make one learning step with the global model. Moreover, each DNN used in FL had much less data and less classes to predict. Because the task was easier, the performance was better. To really compare the centralized learning and FL, we should have us the same dataset.

Conclusion

In this project, we implemented a DNN to recognize and classify attacks in a connected vehicular network. Our proposed model reached 40 % of accuracy with only 50 epochs, though it could have been trained longer.

To improve our project, we can compare our model with one trained on Federated Learning, using the same dataset.