Malware classification using Convolutional Neural Networks

Elaborato del corso Software Security

Christian Marescalco M63001367

Malware Analysis

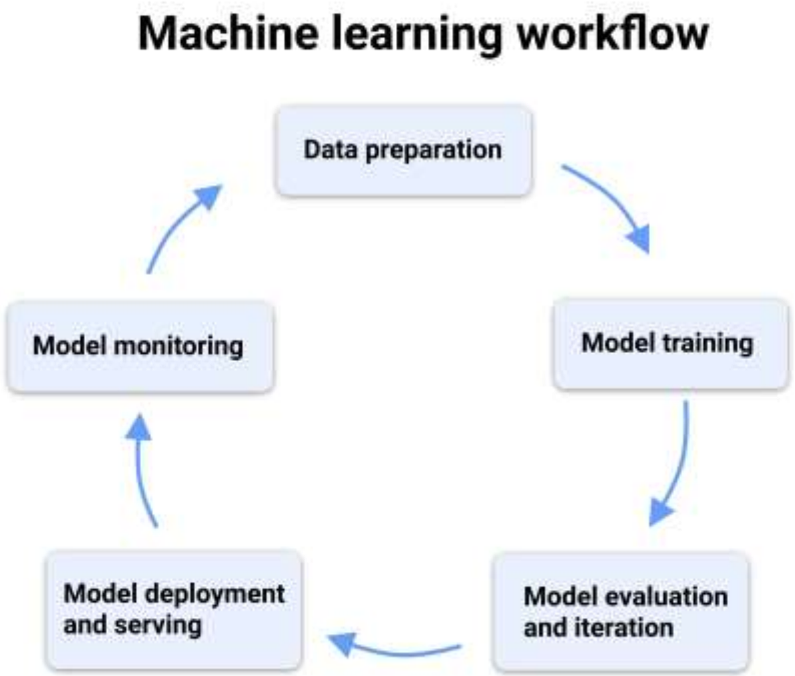
- The process of *dissecting* a malware to understand how it works and how to identify it.
- Classical approaches for extracting features in malware analysis:
  - Static analysis, code or structure examination without execution of the program;
  - Dynamic analysis, execution of the program and behaviour monitoring;
- A malware signature (*fingerprint*) is a set of features that uniquely distinguishes an executable.
- Standard antivirus solutions rely on signature and/or heuristic/behavioural databases to detect malware programs.

Problems with standard antivirus programs

- With the growth of malware volumes, malware analysts need scalable and automated tools to handle large-scale malware samples.
- Malware authors continuosly adapt their techniques to evade detection, for example:
  - Unknown malware variants: an attacker can easily create multiple variants of the same malware.
  - Packed or obfuscated malware: compression and encryption algorithms make the analysis more complicated.
  - Polymorphic malware: the malware uses a polymorphic engine to mutate its features while keeping the same functionality.

ML/DL techniques for Malware Analysis

- Machine learning is well suited for processing large volumes of data.
- It can facilitate the pattern identification and the analysis process.
- The ML/DL workflow has the objective of training a model to solve a task, in this case malware detection/classification.
- There is a preprocessing phase to extract the features from the executables.

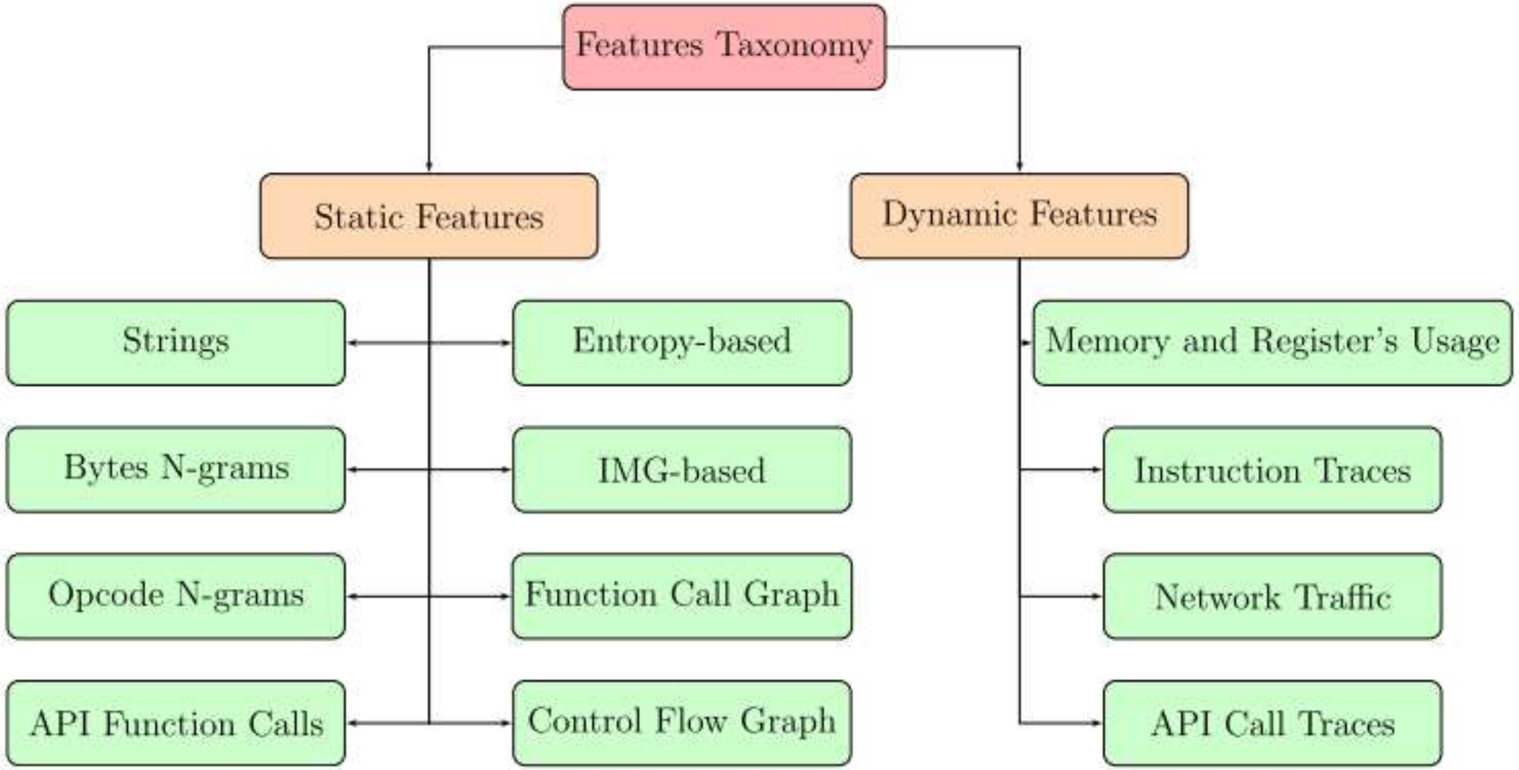


Malware classification and detection

- Malware classification is the process of assigning a malware sample to a specific malware family.
  - In this case, the model outputs the probability of belonging to each malware class for a given executable.
- Malware detection is the process of estabilishing the maliciousness of an executable.
  - The detection model outputs a single probability (*binary model*): malicious or benign.

Executable features used in ML/DL approaches

The feature types can be divided in 2 groups as the malware analysis approaches: static and dynamic features. Features can be combined together to provide a better representation of an executable.



PE Executable files

- Portable executable (PE) is the standard binary file format for executables (.exe) and DLLs (.dll) in Windows.
- It encapsulate all the information necessary for the Windows OS to manage the executable code.
- The header gives info about the external functions used by the program.
- The .text section contains the executable code.
- The .data section contains the global variables.

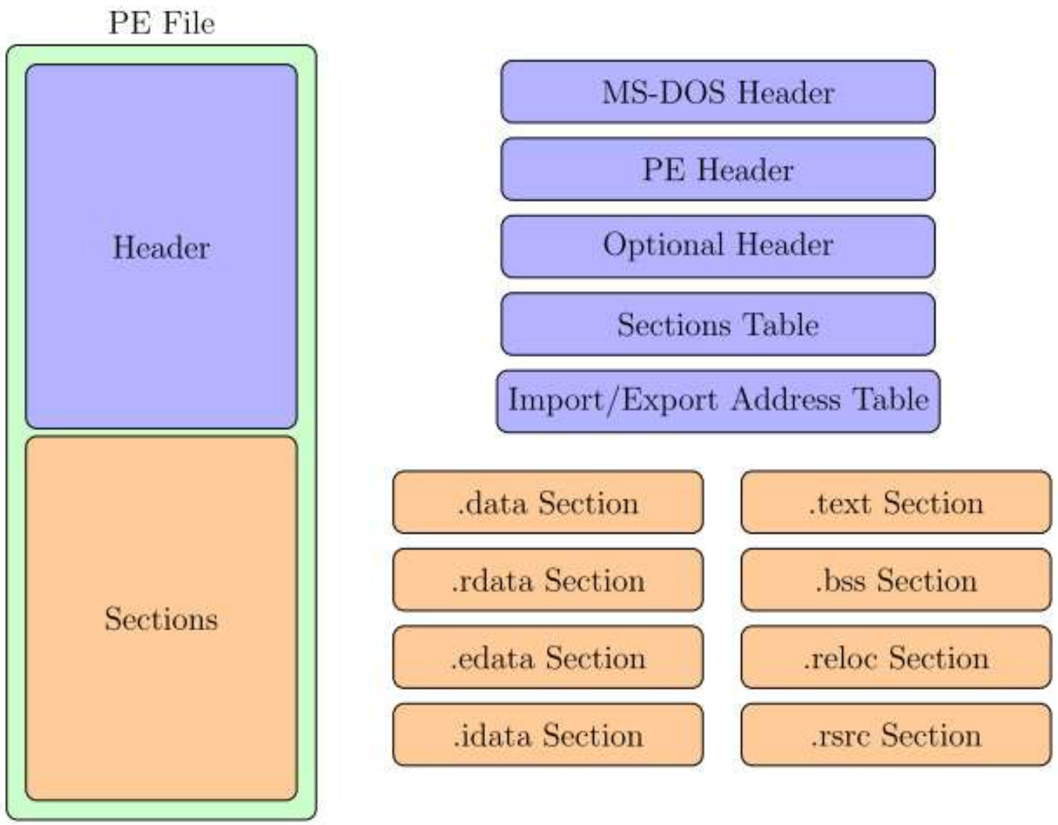
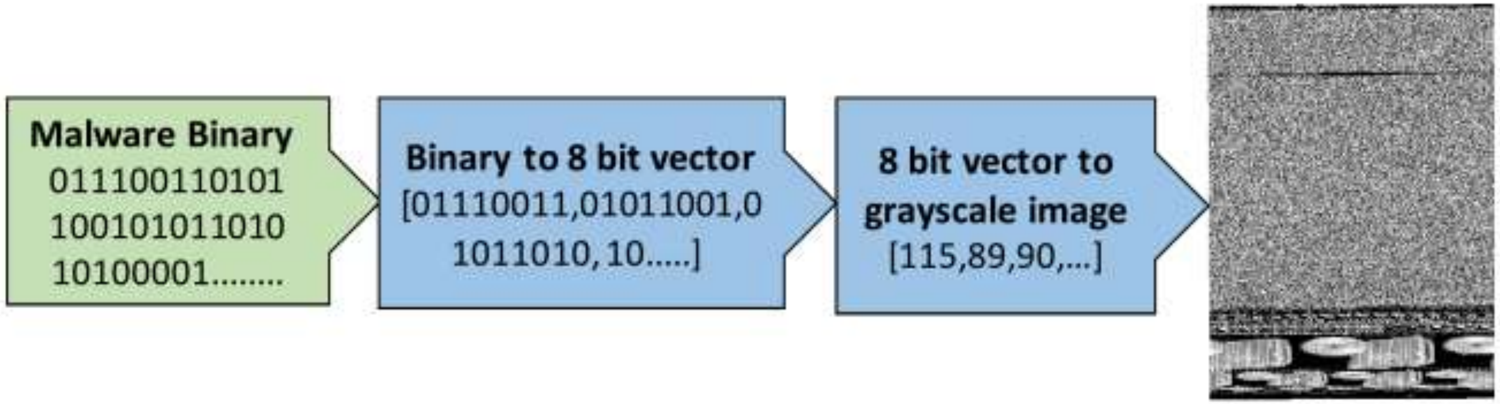


Image-based representation of executables

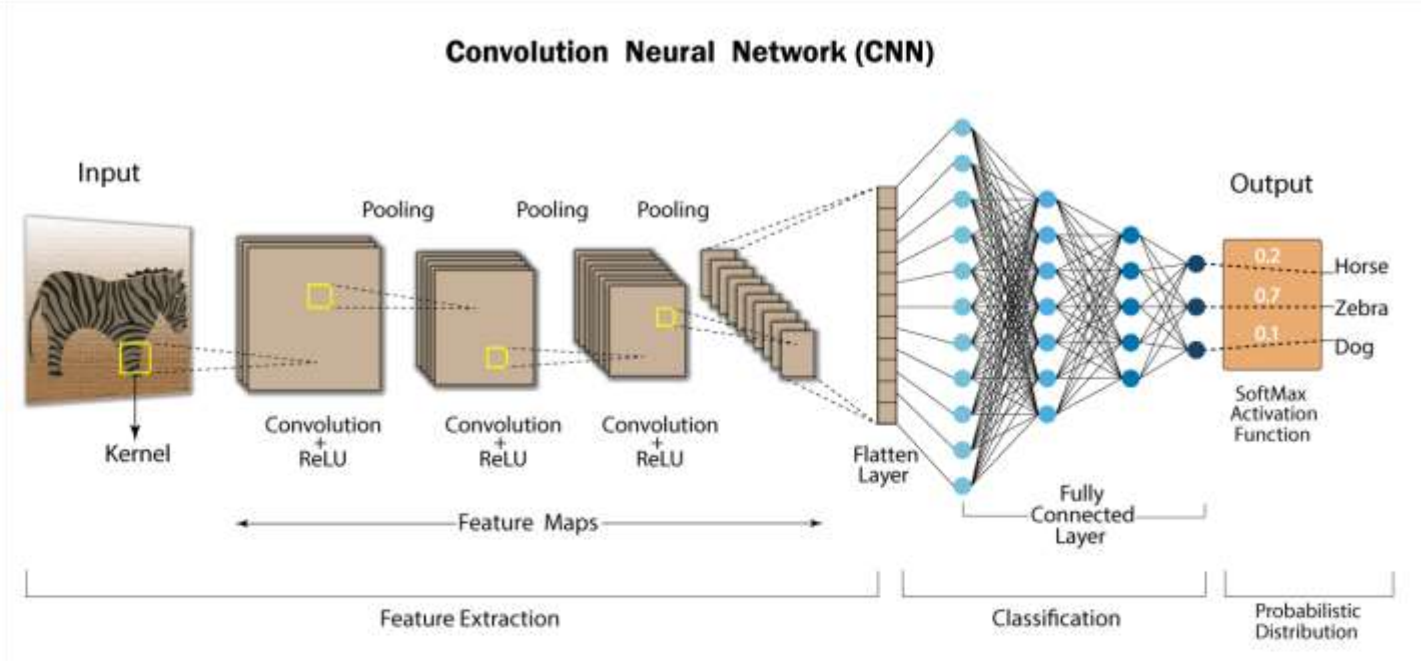
- Each PE executable can be represented as a one-dimensional array of bytes, so with decimal value in the range [0,255].
- The resulting array can be arranged as a 2D array with a reshape to a target image size, obtaining a gray-scale representation of the sample.



Convolutional Neural Networks (CNNs)

- A particular type of Neural Network specifically designed for processing and analyzing images and videos.
- The core component is the *convolutional layer* which uses a moving filter (*kernel*) to detect patterns in the image. The convolutional layer output is a *feature map*.

- The *activation function* is used to indicate the existence of likely features in the input signal.
- *Pooling layers* reduce the spatial size of the feature map in input and provide robustness against noise.
- Fully connected layers combine the learned features and determine a specific target output.



Convolutional Neural Networks (CNNs)

- The CNN approach can be applied for malware classification by using the image representation of executables as input.
- The main advantage of this approach is that **different sections of a binary can be easily separated**.
- To produce new variants, attackers usually change only a small part of the code. So, **re-using old malware to create new binaries has the effect of generating very similar images to the old executable**.
- Additionally, by representing an executable as a gray scale image it is possible to **detect small variations between the samples of the same family**.
- Zero-padding can be easily detected, often used by attackers to reduce the overall entropy.

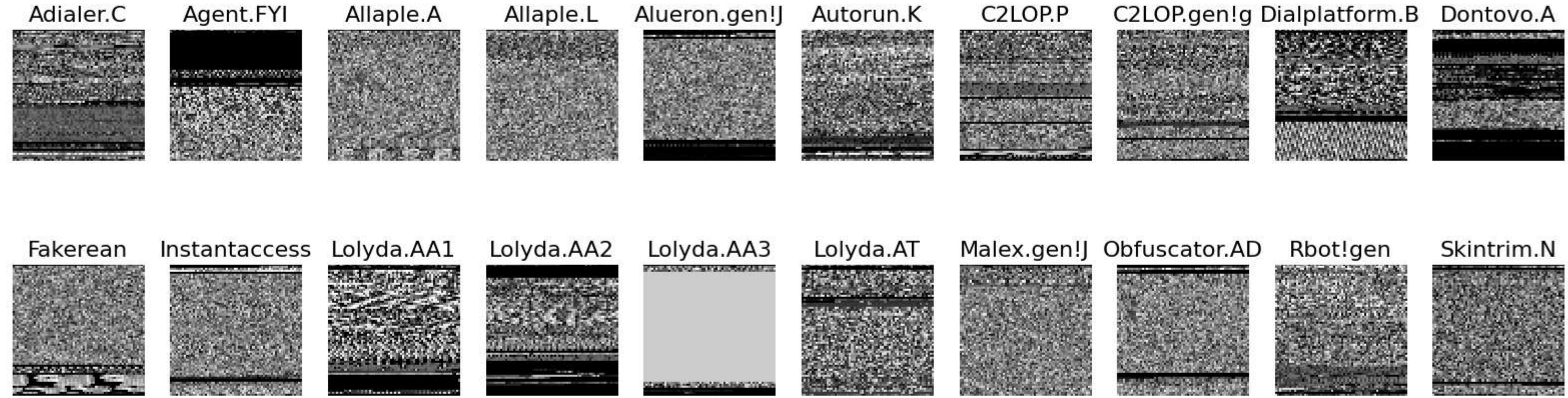
Malimg dataset

- Provided by [Nataraj et al.]
- Consists of 9339 gray scale images of 25 malware classes.
- It contains samples of malicious software packed with UPX: Autorun.K, Malex.gen!J, Rbot!gen, VB.AT, and Yuner.A.
- There are several family variants of the same malware such as Lolyda and Allaple.

No.	Type	Malware family	# di Img
1	Worm	Allaple.L	1591
2	Worm	Allaple.A	2949
3	Worm	Yuner.A	800
4	PWS	lolyda.AA 1	213
5	PWS	lolyda.AA 2	184
6	PWS	lolyda.AA 3	123
7	Trojan	C2Lop.P	146
8	Trojan	C2Lop.gen!G	200
9	Dialer	Instantaccess	431
10	Trojan Downloader	Swizzor.gen!I	132
11	Trojan Downloader	Swizzor.gen!E	128
12	Worm	VB.AT	408
13	Rogue	Fakerean	381
14	Trojan	Alueron.gen!J	198
15	Trojan	Malex.gen!J	136
16	PWS	Lolyda.AT	159
17	Dialer	Adialer.C	125
18	Trojan Downloader	Wintrim.BX	97
19	Dialer	Dialplatform.B	177
20	Trojan Downloader	Dontovo.A	162
21	Trojan Downloader	Obfuscator.AD	142
22	Backdoor	Agent.FYI	116
23	Worm:AutoIT	Autorun.K	106
24	Backdoor	Rbot!gen	158
25	Trojan	Skintrim.N	80

Malimg dataset

- Dataset samples for each class:



- Images from a given family are visually similar, while distinct from those belonging to a different family.

Malimg class examples

- **Dontovo.A class**: is a trojan that downloads and executes arbitrary files.
- Installation:
  - When executed Win32/Dontovo.A runs a copy of %Windows%\svchost.exe and injects code into it.
  - It then deletes its executable.
  - Process injection MITRE ATT&CK T1055.
- Payload:
  - Through svchost.exe, the process contacts the following domain (or others) for configuration data: iframr.com.
  - Downloaded files are saved to the %temp% directory and executed.

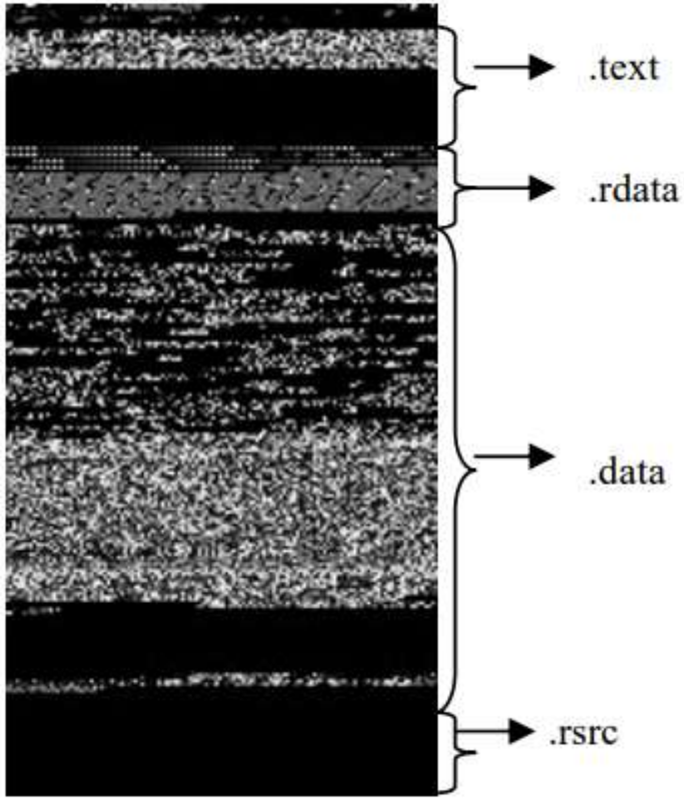
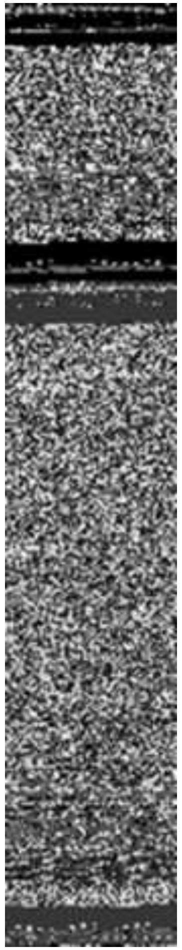


Fig. 2 Various Sections of Trojan: Dontovo.A

https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=TrojanDownloader:Win32/Dontovo.A&threatId=-2147342037

Malimg class examples

- **Lolyda.AT class**: is from a family of trojans that steals account information from popular online games and sends it to a remote server.
- It can also take screenshots, terminate processes, and hook certain APIs.
- Installation:
  - when executed, PWS:Win32/Lolyda.AT drops a DLL with a randomly-generated file name into the Windows system folder.
  - It then modifies the registry to ensure that it is loaded by the 'explorer.exe' process.
  - Modify Registry MITRE ATT&CK T1112, DLL injection MITRE ATT&CK T1055.001.
- Payload:
  - searches the running process memory of several online games to find usernames, passwords, server addresses and characters information.
  - Periodically checks if the foreground window title has the following strings: ACDSee, Internet Explorer. If found, it takes a screenshot and saves it in Windows temporary folder.
  - Hooks APIs, preventing the normal communication between the game client and the game server.



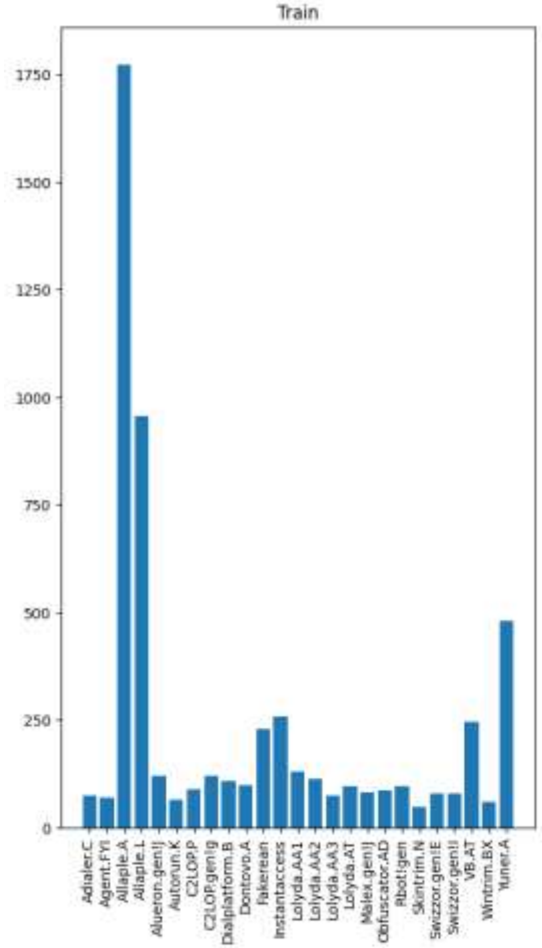
Lolyda.AT sample

https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=PWS:Win32/Lolyda.AT&ThreatID=2147627867

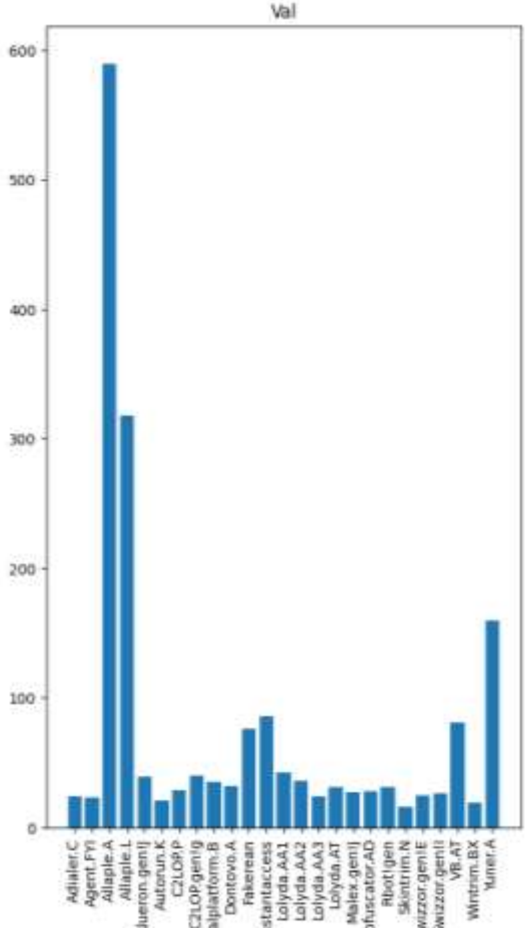
Dataset splitting

- Data partition applied for each class is the following:

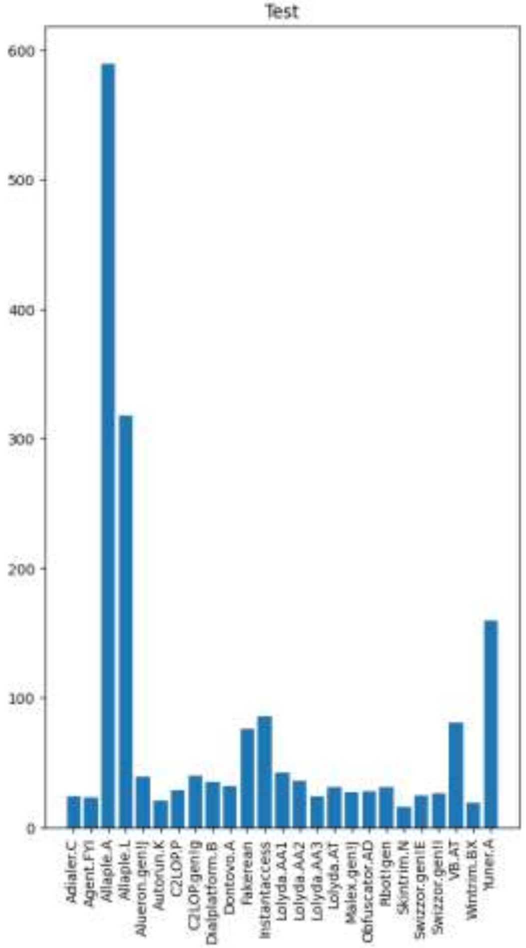
60% for training



20% for validation



20% for test

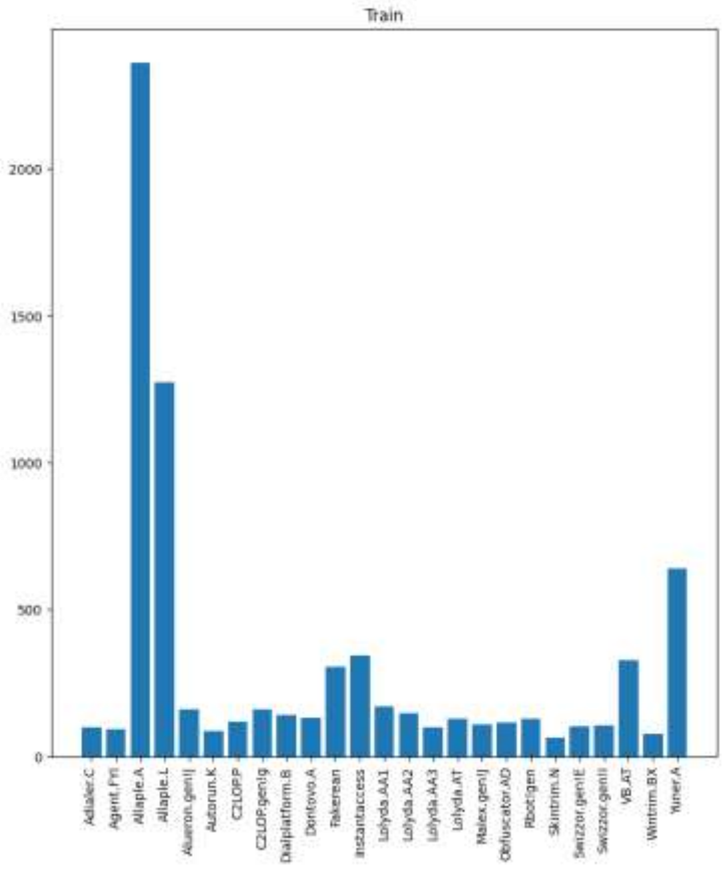


See split_dataset.ipynb for details.

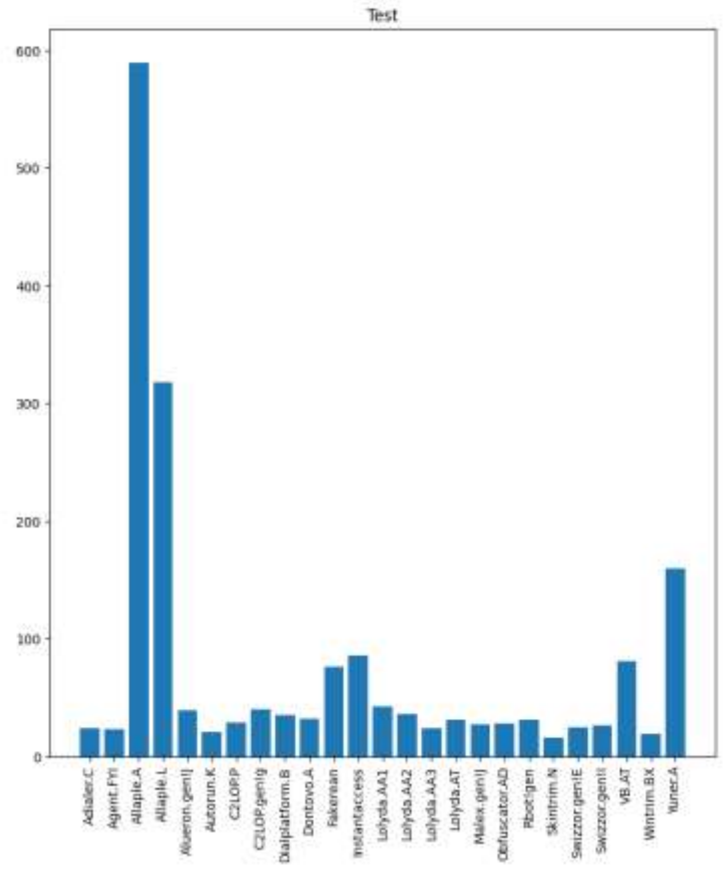
Dataset splitting

- After the best model has been found, validation and training sets can be merged:

80% for training



20% for test



See split_dataset.ipynb for details.

Building the CNN

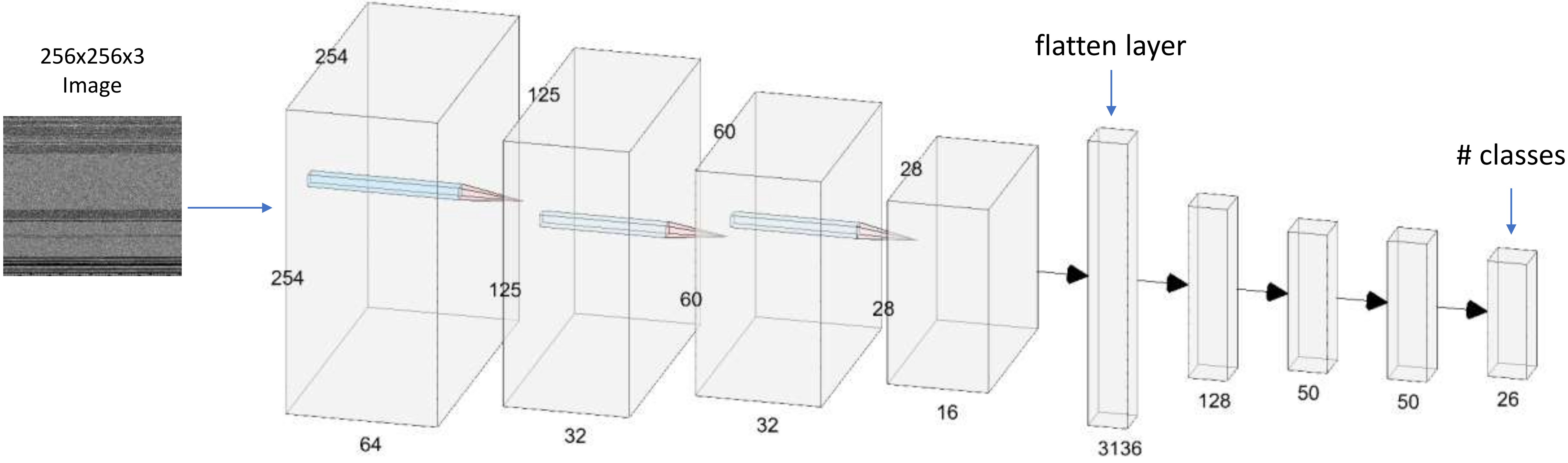
- Tensorflow and Keras were the frameworks used for building and training the CNN.
- All the experiments were executed on Google Colab.

```python
def malware_model():
    Malware_model = Sequential()
    Malware_model.add(Conv2D(64, kernel_size=(3, 3),
                     activation='relu',
                     input_shape=(target_size_custom[0],target_size_custom[1],3)))

    Malware_model.add(MaxPooling2D(pool_size=(2, 2)))
    Malware_model.add(Conv2D(32, kernel_size=(3, 3),
                     activation='relu',
                     input_shape=(target_size_custom[0]//2,target_size_custom[1]//2,3)))

    Malware_model.add(MaxPooling2D(pool_size=(2, 2)))
    Malware_model.add(Conv2D(32, kernel_size=(3, 3),
                     activation='relu',
                     input_shape=(target_size_custom[0]//4,target_size_custom[1]//4,3)))

    Malware_model.add(MaxPooling2D(pool_size=(2, 2)))
    Malware_model.add(Conv2D(16, (3, 3), activation='relu'))
    Malware_model.add(MaxPooling2D(pool_size=(2, 2)))
    Malware_model.add(Dropout(0.25))
    Malware_model.add(Flatten())
    Malware_model.add(Dense(128, activation='relu'))
    Malware_model.add(Dropout(0.25))
    Malware_model.add(Dense(50, activation='relu'))
    Malware_model.add(Dropout(0.5))
    Malware_model.add(Dense(num_classes, activation='softmax'))
    Malware_model.compile(loss='categorical_crossentropy', optimizer = 'adam', metrics=["accuracy"], weighted_metrics=['accuracy'])
    return Malware_model
```

```
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 254, 254, 64)      1792

 max_pooling2d (MaxPooling2D  (None, 127, 127, 64)     0
 )

 conv2d_1 (Conv2D)           (None, 125, 125, 32)      18464

 max_pooling2d_1 (MaxPooling  (None, 62, 62, 32)       0
 2D)

 conv2d_2 (Conv2D)           (None, 60, 60, 32)        9248

 max_pooling2d_2 (MaxPooling  (None, 30, 30, 32)       0
 2D)

 conv2d_3 (Conv2D)           (None, 28, 28, 16)        4624

 max_pooling2d_3 (MaxPooling  (None, 14, 14, 16)       0
 2D)

 dropout (Dropout)           (None, 14, 14, 16)        0

 flatten (Flatten)           (None, 3136)              0

 dense (Dense)               (None, 128)               401536

 dropout_1 (Dropout)         (None, 128)               0

 dense_1 (Dense)             (None, 50)                6450

 dropout_2 (Dropout)         (None, 50)                0

 dense_2 (Dense)             (None, 26)                1326

=================================================================
Total params: 443,440
Trainable params: 443,440
Non-trainable params: 0
```

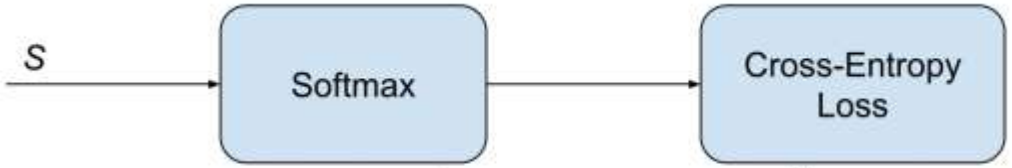
Building the CNN

- Only trainable convolutional layers are showed.
- Between each of the 2D layer, there is a max pooling layer and a dropout layer.
- Between each dense layer, there is a dropout layer.



Building the CNN

- Loss function:
  - is a mathematical function that measures the discrepancy between the predicted output of a model and the true or expected output.
  - The choice of the loss function depends on the specific problem and the nature of the data.
  - In this case, a multi-class classification problem, the *Categorical Cross-Entropy* (Softmax loss) is used:



$$f(s)_i = \frac{e^{s_i}}{\sum_j^C e^{s_j}} \qquad CE = -\sum_i^C t_i log(f(s)_i)$$

- i and j iterate through classes;
- C is the number of classes;
- s is the prediction vector;
- t is the ground truth vector;

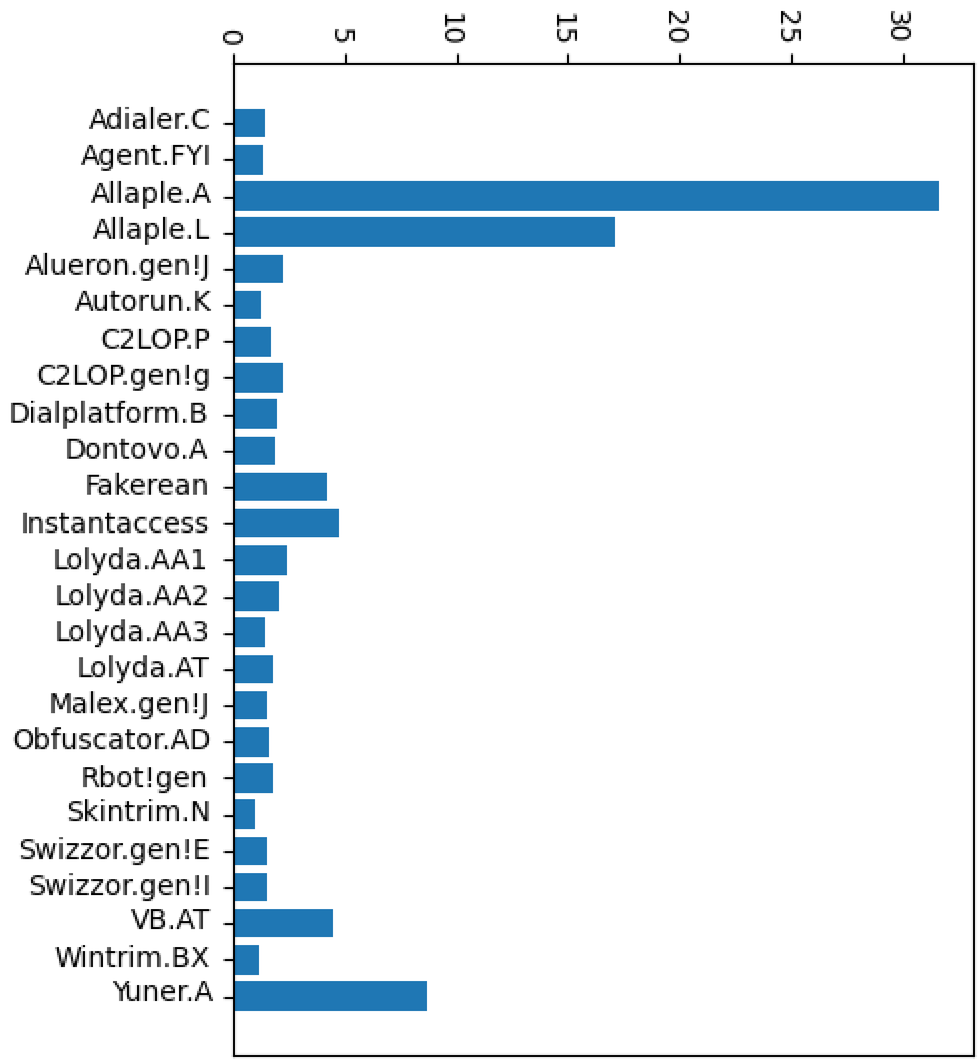
- *Optimizer Adam*:
  - It stands for "Adaptive Moment Estimation“
  - It is an adaptive optimization algorithm commonly used in training deep learning models.

Hyperparameters tuning

- Batch size:
  - indicates the number of training examples used in one iteration of the training process.
  - Trade-off between larger batch-size (faster training time) and small batch-size (better model generalization).
  - In this case a batch size = 32 is chosen.
- Target image size:
  - Refers to the image given in input to the CNN.
  - In this case 256x256 pixels is chosen.
- Learning rate (LR):
  - Determines the step size at which the model updates its parameters during the training.
  - If the LR is too high the model, the model may fail to converge, otherwise if it is too low the model will slow down the convergence.
  - In this case a LR = 0.001 is chosen.

Analysis of class distribution

- Unbalanced dataset:
  - Allaple.A has the majority of samples, more than 30%.
  - Other classes also have an high percentage of samples.
- This issue, if not dealt correctly, will bias the model towards high frequency classes.

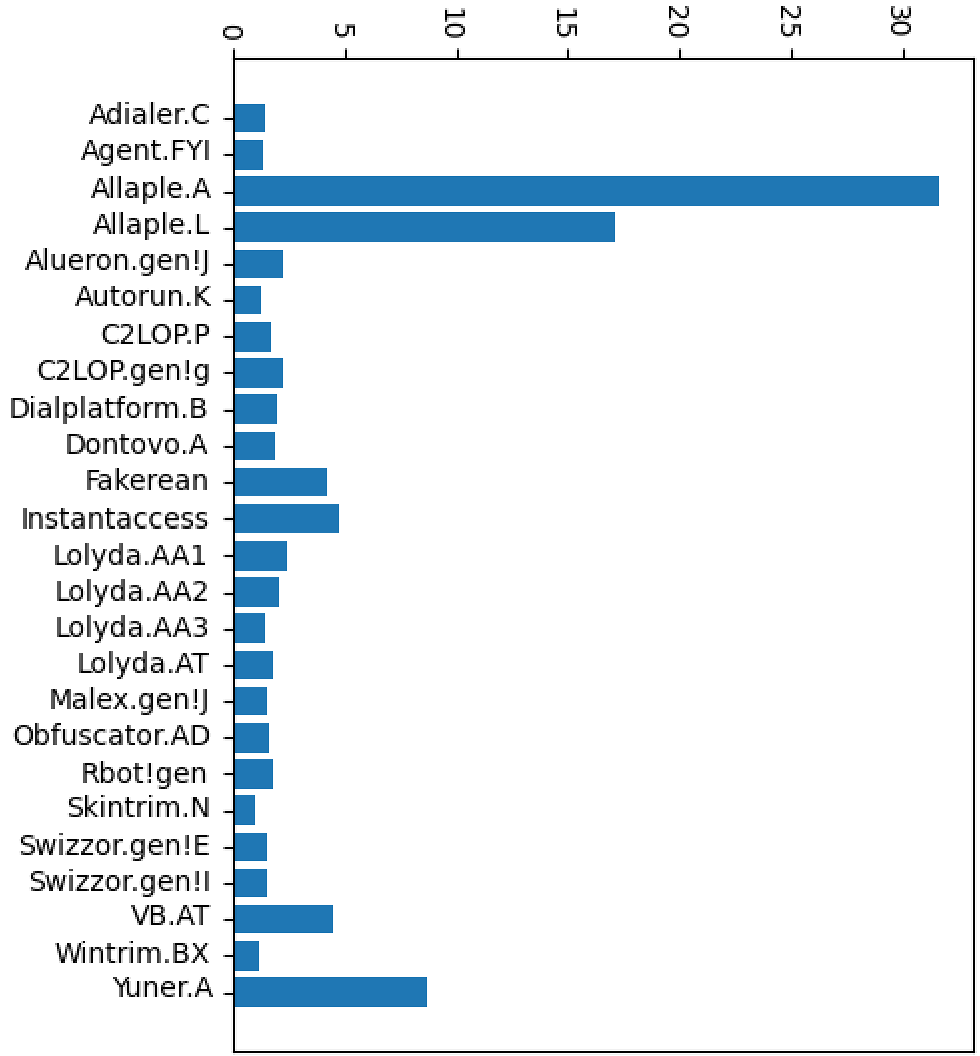


Analysis of class distribution

- Class weights calculation based on number of samples:

$$\omega_i = \frac{\#samples}{\#\text{classes} \cdot n_i}$$

- ω_i represents the i-th class weight.
- n_i is the number of occurrences of the i-th class.
- Assigning a lower weight to popular classes helps the model to better perform the training.



Class weight calculation

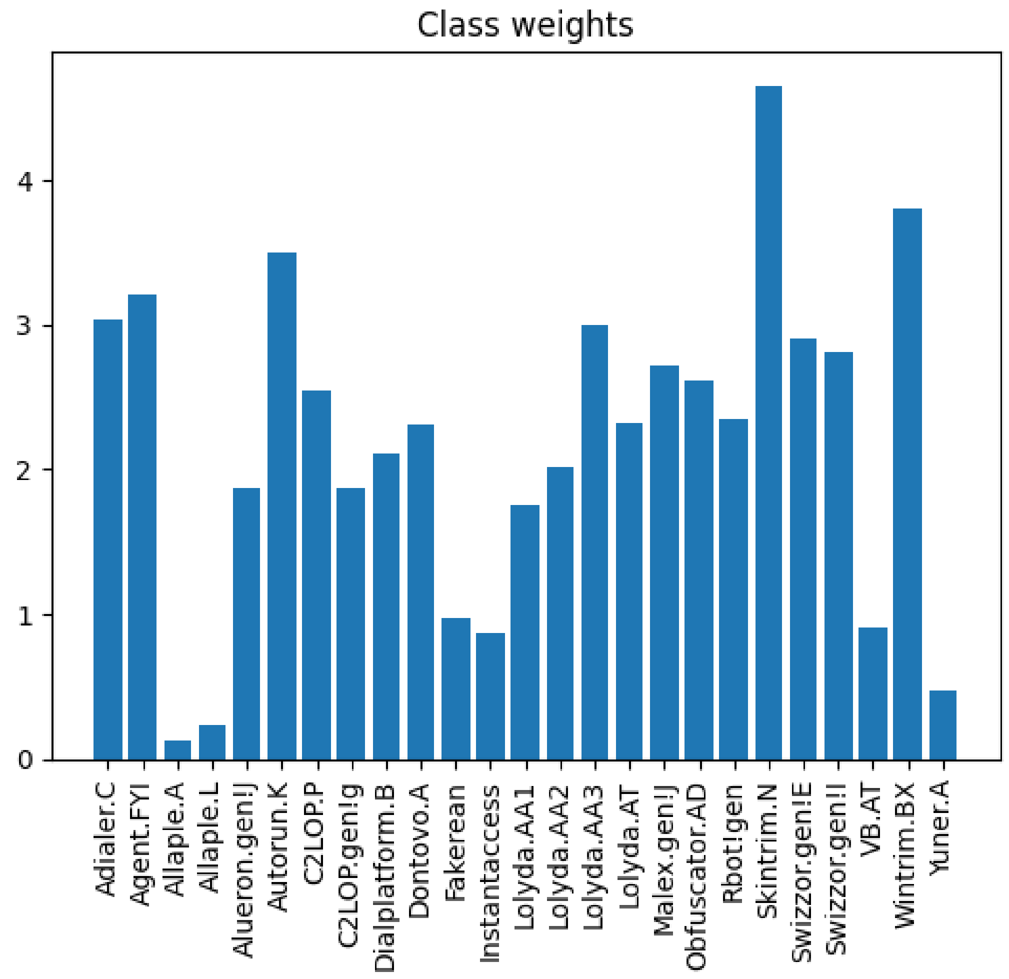
- Computing the weight of each malware class with the sklearn library.

Class weight calculation

```
[ ]  train_labels = train_df.replace({"target":class_index})['target'].to_numpy()
     class_indices =np.array(list(class_index.values()))
     class_indices
```

```
[ ]  from sklearn.utils import class_weight
     class_weights = class_weight.compute_class_weight(class_weight = 'balanced',
                                                       classes = class_indices,
                                                       y = train_labels)

     class_weights = dict(zip(np.unique(train_labels), class_weights))
     class_weights
```

```
{0: 3.053469387755102,
 1: 3.2176344086021507,
 2: 0.12679661016949154,
 3: 0.23506677140612725,
 4: 1.8820125786163522,
 5: 3.5204705882352942,
 6: 2.5576068376068375,
 7: 1.87025,
 8: 2.1073239436619717,
```



Evaluation Metrics

The following metrics are applied for each class:

- *Precision*: $P = \dfrac{T_p}{T_p + F_p}$
- *F1 Score*: $F_1 = 2 \cdot \dfrac{P \cdot R}{P + R}$
- *Recall*: $R = \dfrac{T_p}{T_p + Fn}$
- *Accuracy:* $A = \dfrac{T_p + T_n}{T_p + T_n + F_p + F_n}$

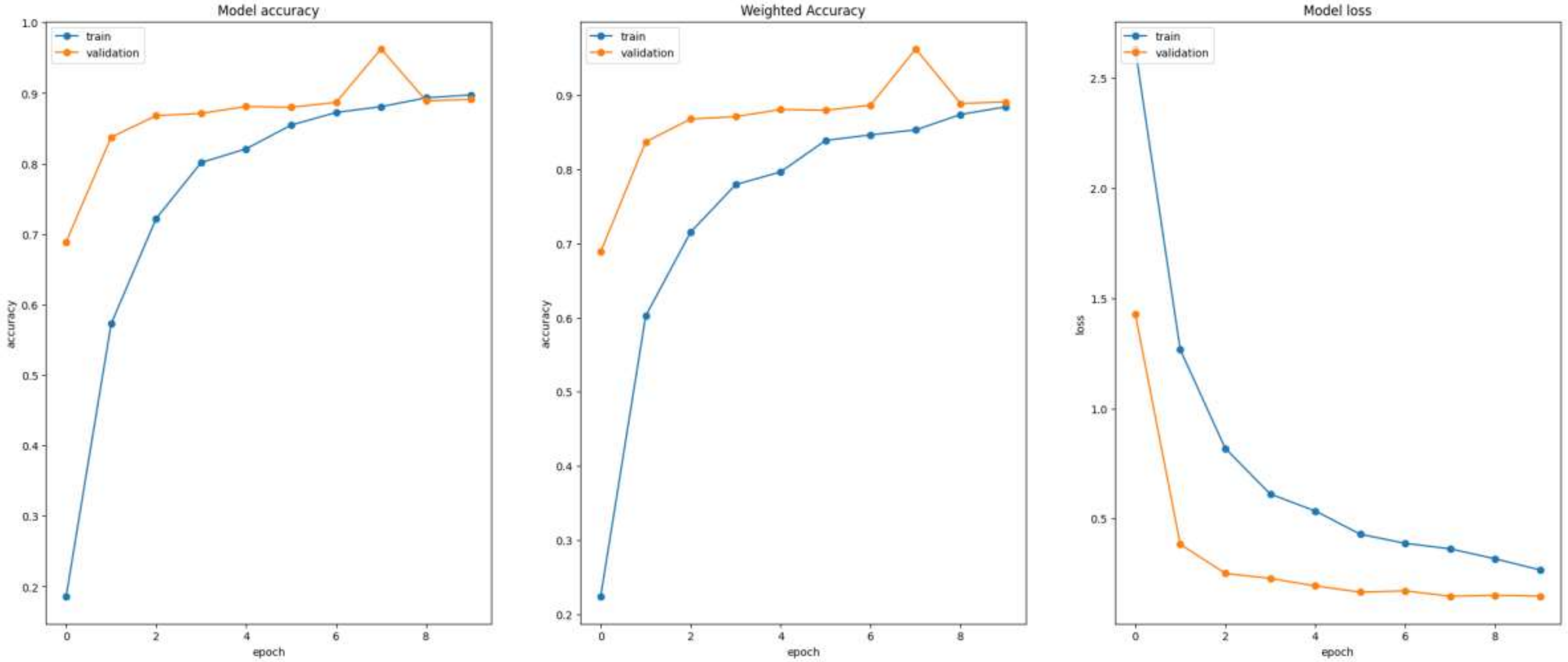
- Then, the average of the individual metrics is calculated, obtaining:
  - *macro_precision*, *macro_recall*, *macro_f1-score, avg_accuracy;*
- and the weighted average of the metrics:
  - *weighted_precision*, *weighted_recall*, *weighted_f1-score, weighted_accuracy.*

Training phase on Malimg dataset

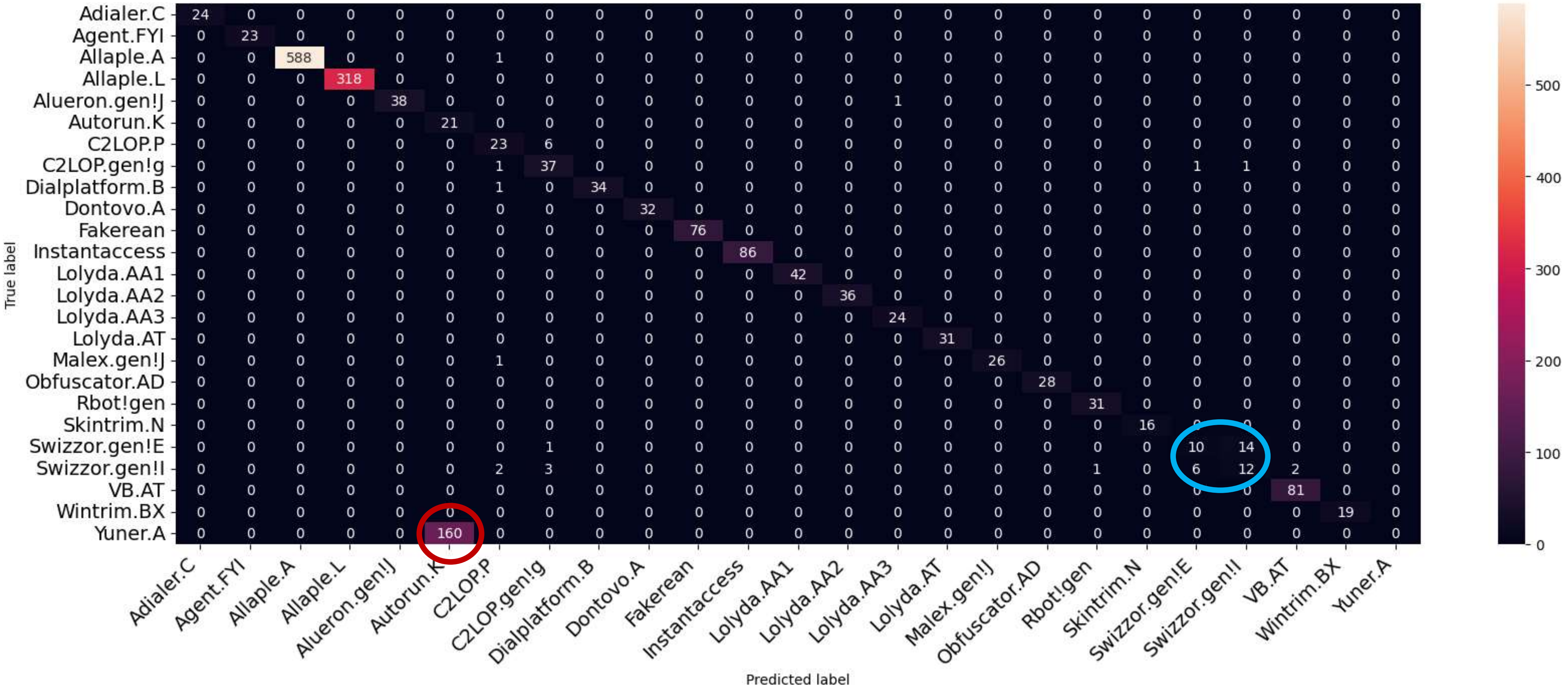
- Training on 10 epochs.
- Using the validation data.
- Using the class weights to balance the dataset.
- To evaluate the quality of training different metrics are plotted:
  - Training loss and validation loss;
  - Accuracy and validation accuracy;
  - Weighted accuracy and validation weighted accuracy;

- Average metrics obtained on the validation set (1858 samples):
  - loss: 0.1476 – val_accuracy: 0.8913 – val_weighted_accuracy: 0.8913

Training phase on Malimg dataset



Evaluation phase on validation set



Evaluation phase on validation set

- A lot of misclassification to Autorun.K on the Yuner.A samples.
- Other training experiments showed the inverse misclassification.

- Misclassification between malware samples of same family but different classes (*variants*): Swizzor.gen!E and Swizzor.gen!I.
- This is probably caused by a similarity between the two variants.

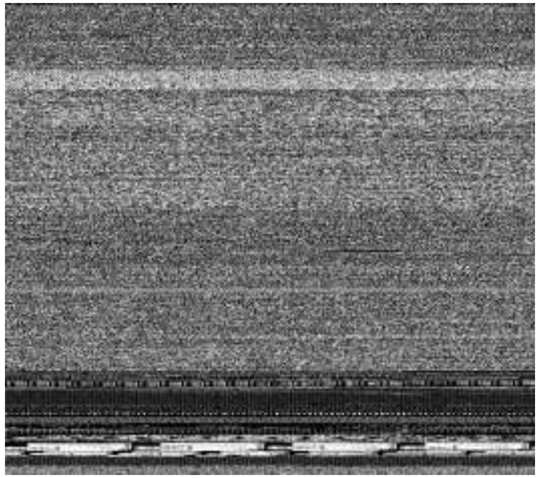
- Other samples are classified almost correctly although the different variants of each family.
- Average F1-Score, Precision and Recall can be improved by resolving the first misclassification.

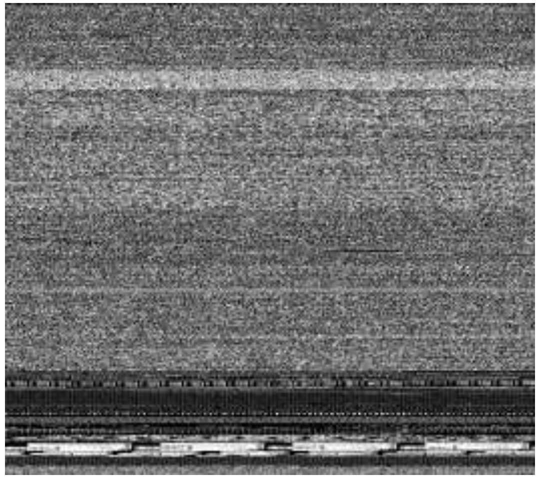
Class	precision	recall	f1-score	support
Adialer.C	1.0	1.0	1.0	24.0
Agent.FYI	1.0	1.0	1.0	23.0
Allaple.A	1.0	0.9983	0.99915	589.0
Allaple.L	1.0	1.0	1.0	318.0
Alueron.gen!J	1.0	0.97436	0.98701	39.0
Autorun.K	0.11602	1.0	0.20792	21.0
C2LOP.P	0.7931	0.7931	0.7931	29.0
C2LOP.gen!g	0.78723	0.925	0.85057	40.0
Dialplatform.B	1.0	0.97143	0.98551	35.0
Dontovo.A	1.0	1.0	1.0	32.0
Fakerean	1.0	1.0	1.0	76.0
Instantaccess	1.0	1.0	1.0	86.0
Lolyda.AA1	1.0	1.0	1.0	42.0
Lolyda.AA2	1.0	1.0	1.0	36.0
Lolyda.AA3	0.96	1.0	0.97959	24.0
Lolyda.AT	1.0	1.0	1.0	31.0
Malex.gen!J	1.0	0.96296	0.98113	27.0
Obfuscator.AD	1.0	1.0	1.0	28.0
Rbot!gen	0.96875	1.0	0.98413	31.0
Skintrim.N	1.0	1.0	1.0	16.0
Swizzor.gen!E	0.58824	0.4	0.47619	25.0
Swizzor.gen!I	0.44444	0.46154	0.45283	26.0
VB.AT	0.9759	1.0	0.9878	81.0
Wintrim.BX	1.0	1.0	1.0	19.0
Yuner.A	0.0	0.0	0.0	160.0
accuracy	0.89128	0.89128	0.89128	0.89128
macro avg	0.86535	0.89947	0.8674	1858.0
weighted avg	0.88068	0.89128	0.88163	1858.0

Autorun.K and Yuner.A

- Autorun.K class: is from a family of worms that targets the autorun functionality in Windows.
  - The autorun feature is designed to automatically run programs and scripts when removable media (USB, CD) are inserted.
  - When an infected device is connected, the worm uses the autorun feature to execute malicious code and infect other devices.
- Yuner.A class: same family of worms that targets the autorun functionality in Windows.
- Both classes use the same packer UPX.
- By comparing image samples of the 2 classes, there is almost no difference.

Yuner.A





Autorun.K

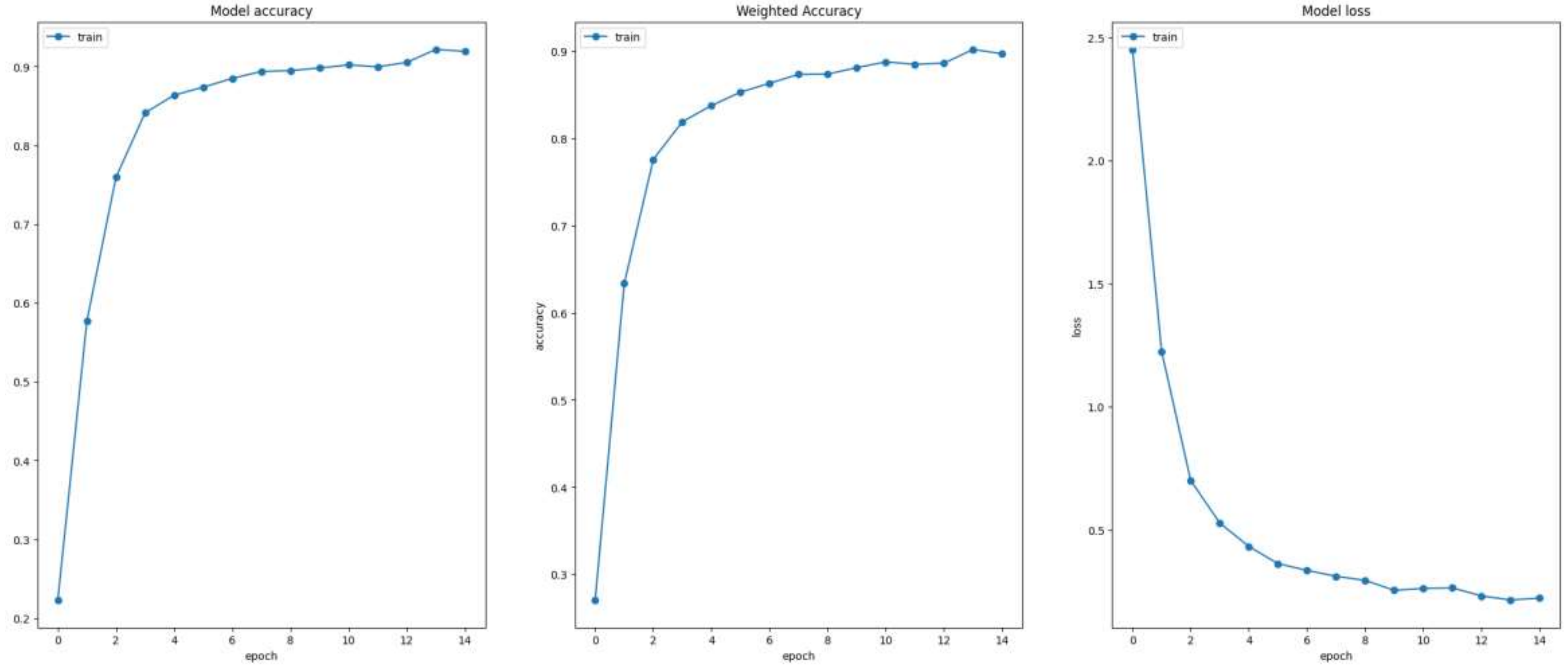
https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Worm:Win32/Yuner.A&ThreatID=2147600986

Retraining phase

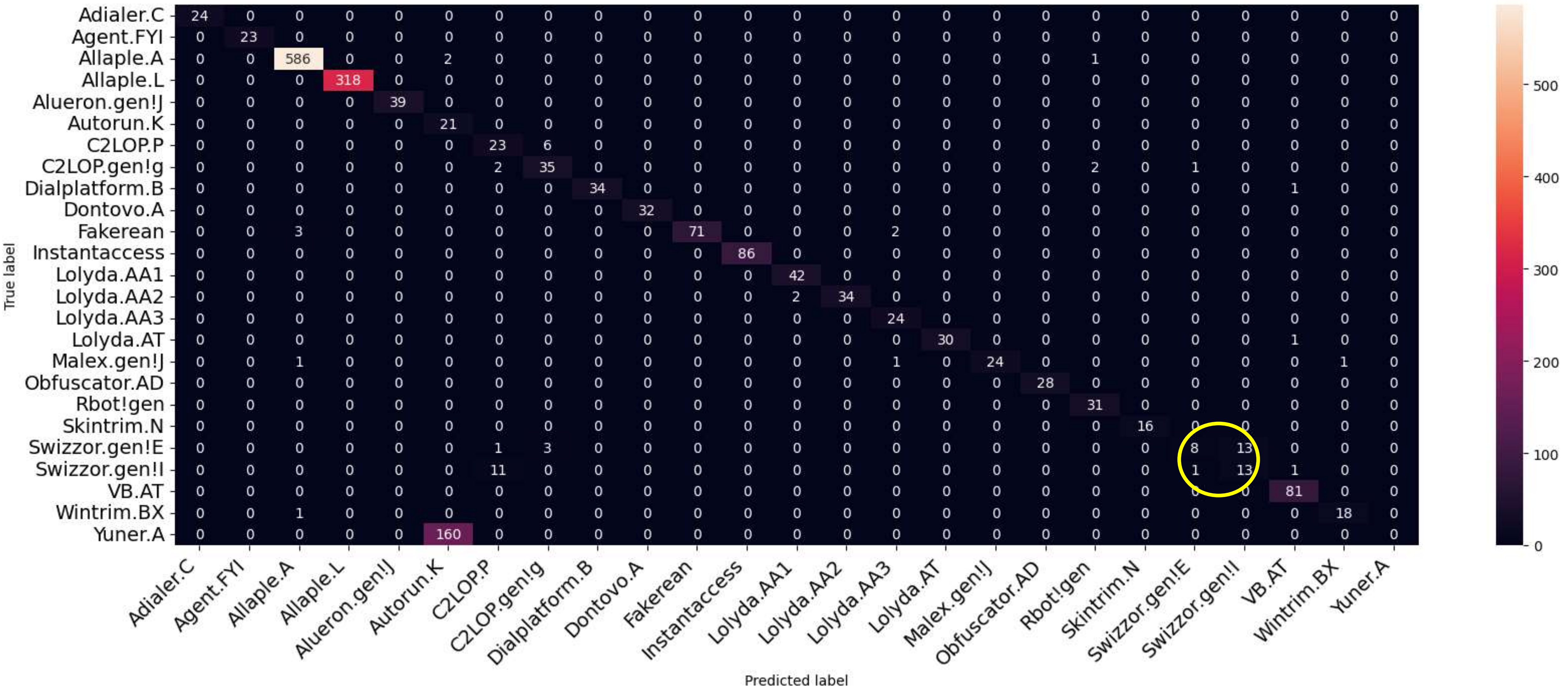
- Training on 15 epochs.
- Merging validation and training data (80%).
- Using the class weights to balance the dataset.
- To evaluate the quality of training different metrics are plotted:
  - Training loss;
  - Accuracy;
  - Weighted accuracy;

- Average metrics obtained on the test set (1858 samples):
  - loss: 0.2229 - accuracy: 0.9193 - weighted_accuracy: 0.8969

Retraining phase



Retraining phase



Retraining phase

- Still a lot of misclassification to Autorun.K on the Yuner.A samples.
- Lower misclassification between malware variants: Swizzor.gen!E and Swizzor.gen!I.
- Other samples are classified almost correctly although the different variants of each family.
- Generally, same performance as the previous training.

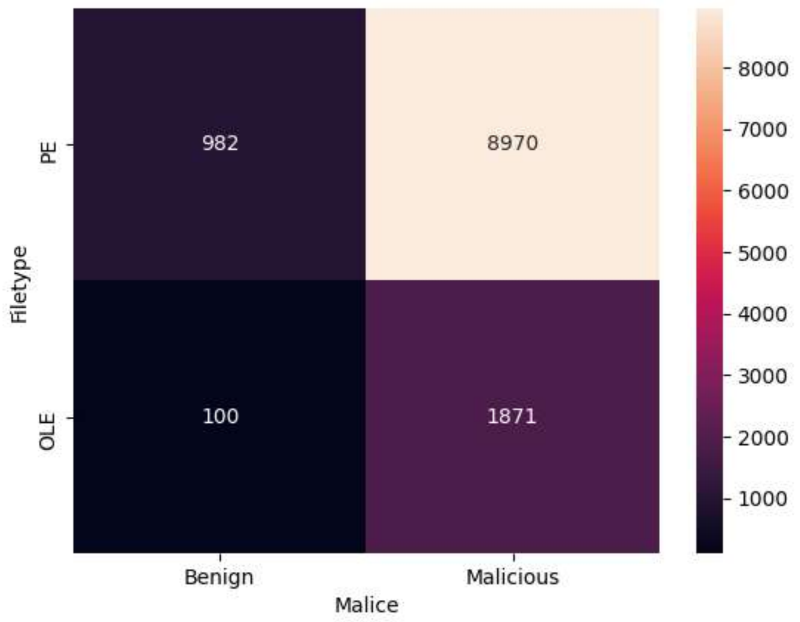
Class	precision	recall	f1-score	support
Adialer.C	1.0	1.0	1.0	24.0
Agent.FYI	1.0	1.0	1.0	23.0
Allaple.A	0.99154	0.99491	0.99322	589.0
Allaple.L	1.0	1.0	1.0	318.0
Alueron.gen!J	1.0	1.0	1.0	39.0
Autorun.K	0.11475	1.0	0.20588	21.0
C2LOP.P	0.62162	0.7931	0.69697	29.0
C2LOP.gen!g	0.79545	0.875	0.83333	40.0
Dialplatform.B	1.0	0.97143	0.98551	35.0
Dontovo.A	1.0	1.0	1.0	32.0
Fakerean	1.0	0.93421	0.96599	76.0
Instantaccess	1.0	1.0	1.0	86.0
Lolyda.AA1	0.95455	1.0	0.97674	42.0
Lolyda.AA2	1.0	0.94444	0.97143	36.0
Lolyda.AA3	0.88889	1.0	0.94118	24.0
Lolyda.AT	1.0	0.96774	0.98361	31.0
Malex.gen!J	1.0	0.88889	0.94118	27.0
Obfuscator.AD	1.0	1.0	1.0	28.0
Rbot!gen	0.91176	1.0	0.95385	31.0
Skintrim.N	1.0	1.0	1.0	16.0
Swizzor.gen!E	0.8	0.32	0.45714	25.0
Swizzor.gen!I	0.5	0.5	0.5	26.0
VB.AT	0.96429	1.0	0.98182	81.0
Wintrim.BX	0.94737	0.94737	0.94737	19.0
Yuner.A	0.0	0.0	0.0	160.0
accuracy	0.88321	0.88321	0.88321	0.88321
macro avg	0.85961	0.88548	0.85341	1858.0
weighted avg	0.87517	0.88321	0.87341	1858.0

Adding a benign class to the Dataset

- Open source dataset:
  - https://github.com/iosifache/DikeDataset/tree/main

- The DikeDataset is a labeled dataset containing benign and malicious PE and OLE files. It includes another dataset for PE files:
  - [Malware Detection PE-Based Analysis Using Deep Learning Algorithm Dataset]

- Benign executable files are taken from installed folders of applications of legitimate software from different categories.
- VirusTotal was used to ensure that each file belongs to the benign class.
- Only PE files (.exe) from the benign class were extracted from the dataset (982 samples).



Adding a benign class to the Dataset

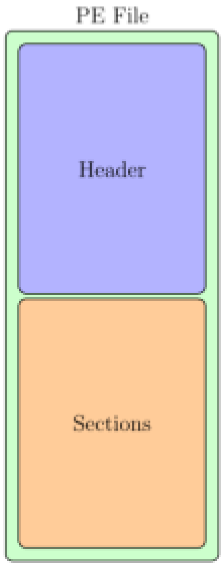
- The benign subset is composed by very diverse executable files, this will help the generalization of the model.

- File examples for the benign data:

    - ApacheMonitor.exe, tomcat7.exe, vmware.exe, xampp_start.exe ecc.
    - Matlab, Octave executables.
    - Microsoft tools: svchost.exe, dos2unix.exe ecc.
    - File packers: 7zip.exe, WinRar.exe.

Adding a benign class to the Dataset

- Each PE executable can be represented as a one-dimensional array of bytes, so with decimal value in the range [0,255]. Then, the array is reshaped as an image.

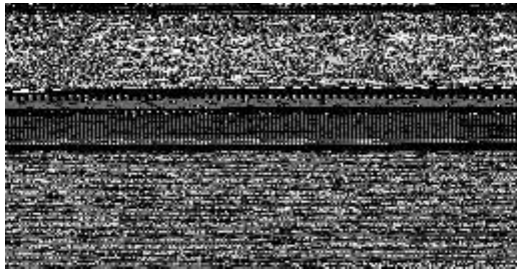


To .bytes

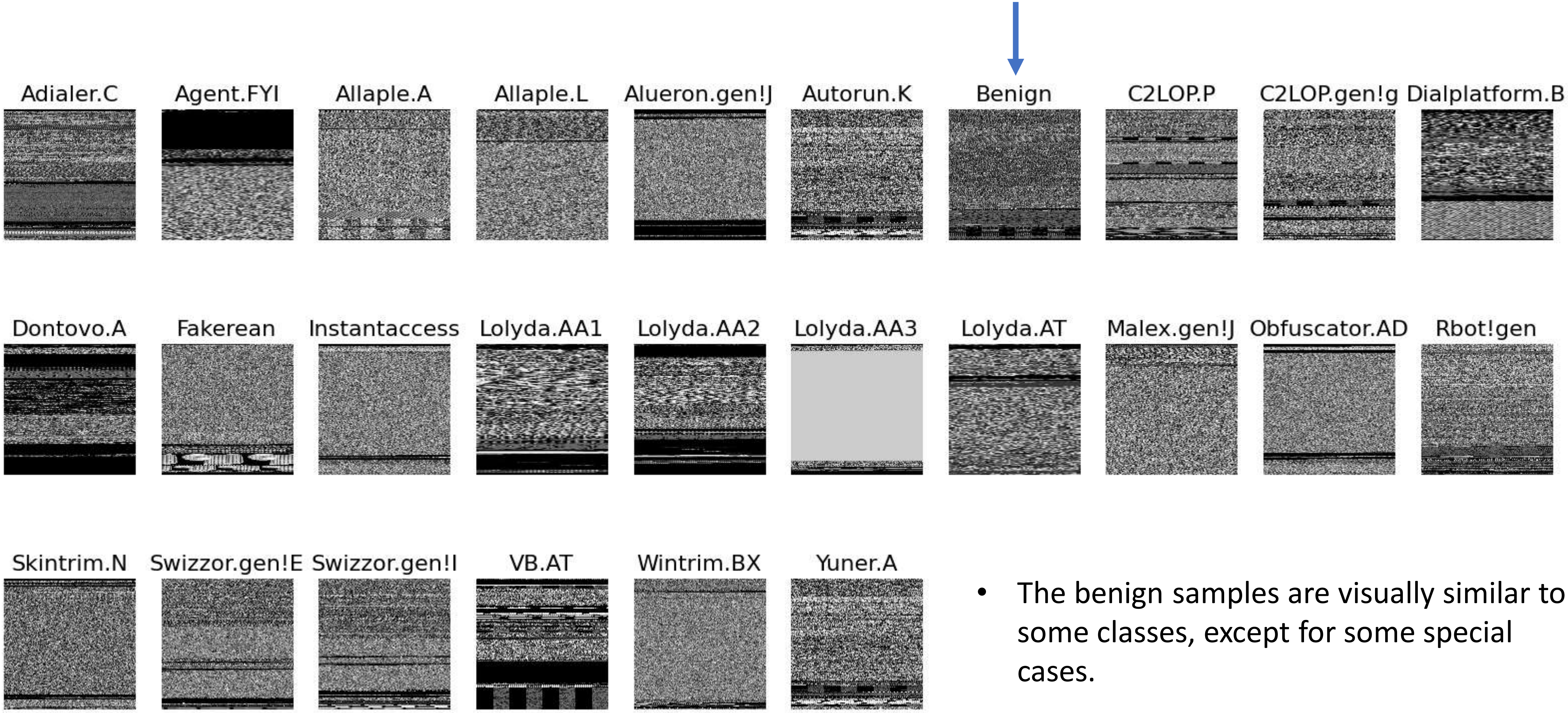

8baaee3f0699d15a860e283880baebca72912df873c5b00fdf4d55c84ad06de7.bytes

```
0x4d 0x5a 0x90 0x00 0x03 0x00 0x00 0x00 0x04 0x00 0x00 0x00 0xff 0xff 0x00 0x00
0xb8 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x40 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0xf0 0x00 0x00 0x00
0x0e 0x1f 0xba 0x0e 0x00 0xb4 0x09 0xcd 0x21 0xb8 0x01 0x4c 0xcd 0x21 0x54 0x68
0x69 0x73 0x20 0x70 0x72 0x6f 0x67 0x72 0x61 0x6d 0x20 0x63 0x61 0x6e 0x6e 0x6f
0x74 0x20 0x62 0x65 0x20 0x72 0x75 0x6e 0x20 0x69 0x6e 0x20 0x44 0x4f 0x53 0x20
0x6d 0x6f 0x64 0x65 0x2e 0x0d 0x0d 0x0a 0x24 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xa1 0x8e 0xae 0x72 0xe5 0xef 0xc0 0x21 0xe5 0xef 0xc0 0x21 0xe5 0xef 0xc0 0x21
0x19 0x98 0x7d 0x21 0xe4 0xef 0xc0 0x21 0x19 0x98 0x7b 0x21 0xe4 0xef 0xc0 0x21
0x19 0x98 0x7c 0x21 0xf6 0xef 0xc0 0x21 0x19 0x98 0x79 0x21 0xec 0xef 0xc0 0x21
0xe5 0xef 0xc1 0x21 0xa1 0xef 0xc0 0x21 0x19 0x98 0x01 0x21 0xe2 0xef 0xc0 0x21
0x19 0x98 0x77 0x21 0xe4 0xef 0xc0 0x21 0x19 0x98 0x7a 0x21 0xe4 0xef 0xc0 0x21
0x52 0x69 0x63 0x68 0xe5 0xef 0xc0 0x21 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x50 0x45 0x00 0x00 0x4c 0x01 0x05 0x00 0xfb 0xa8 0x10 0x50 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0xe0 0x00 0x02 0x01 0x0b 0x01 0x0a 0x0a 0x00 0x28 0x00 0x00
0x00 0x20 0x00 0x00 0x00 0x00 0x00 0x00 0x6c 0x31 0x00 0x00 0x00 0x10 0x00 0x00
```

To .png



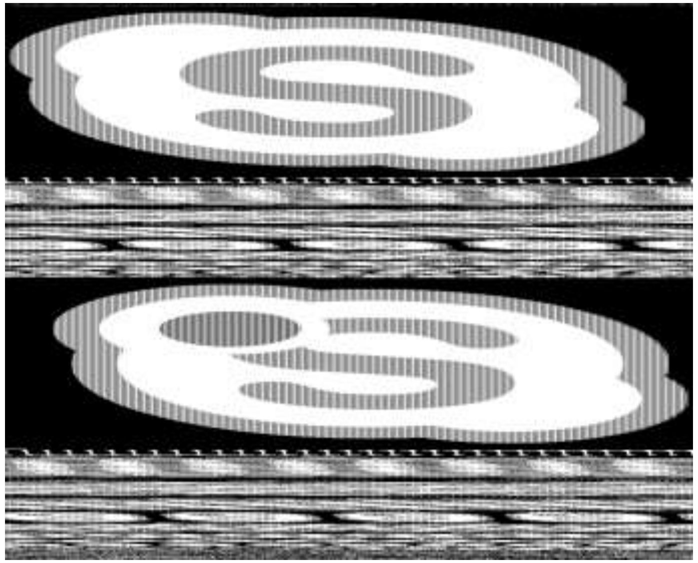
Adding a benign class to the Dataset

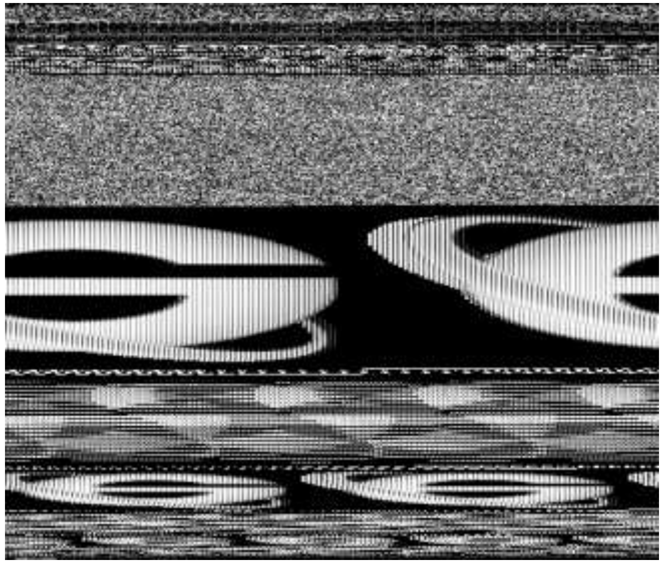


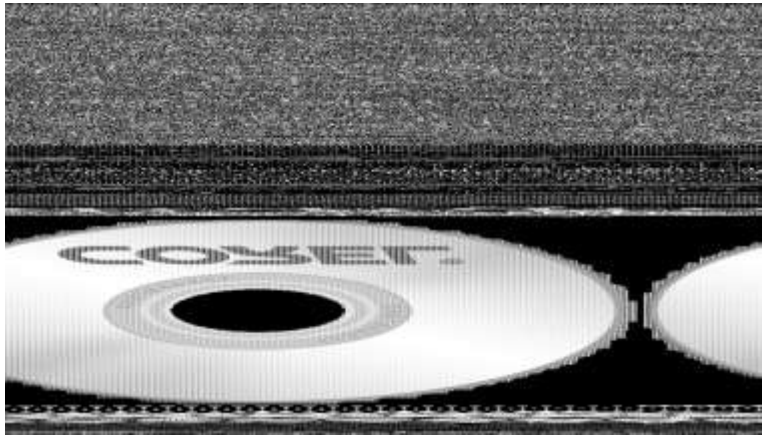
- The benign samples are visually similar to some classes, except for some special cases.

Adding a benign class to the Dataset

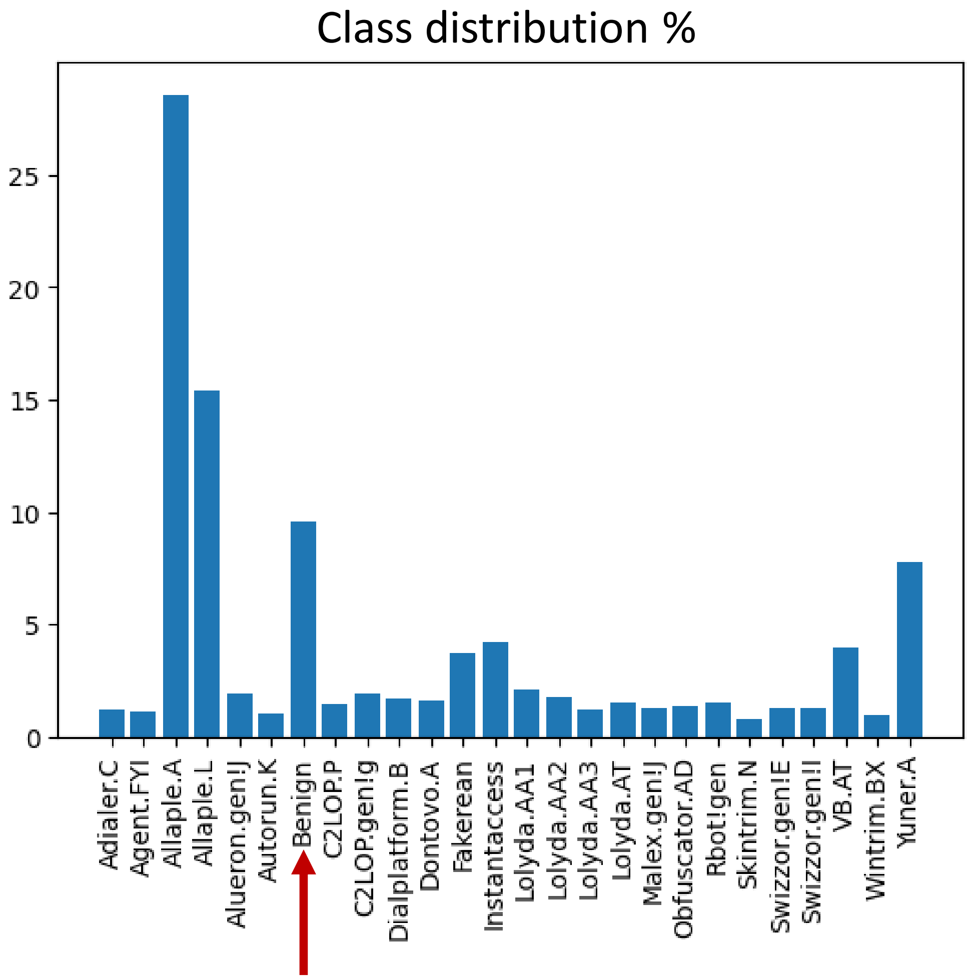
- The benign samples are visually similar to some classes, except for some special cases where they contain a logo in their resources section.

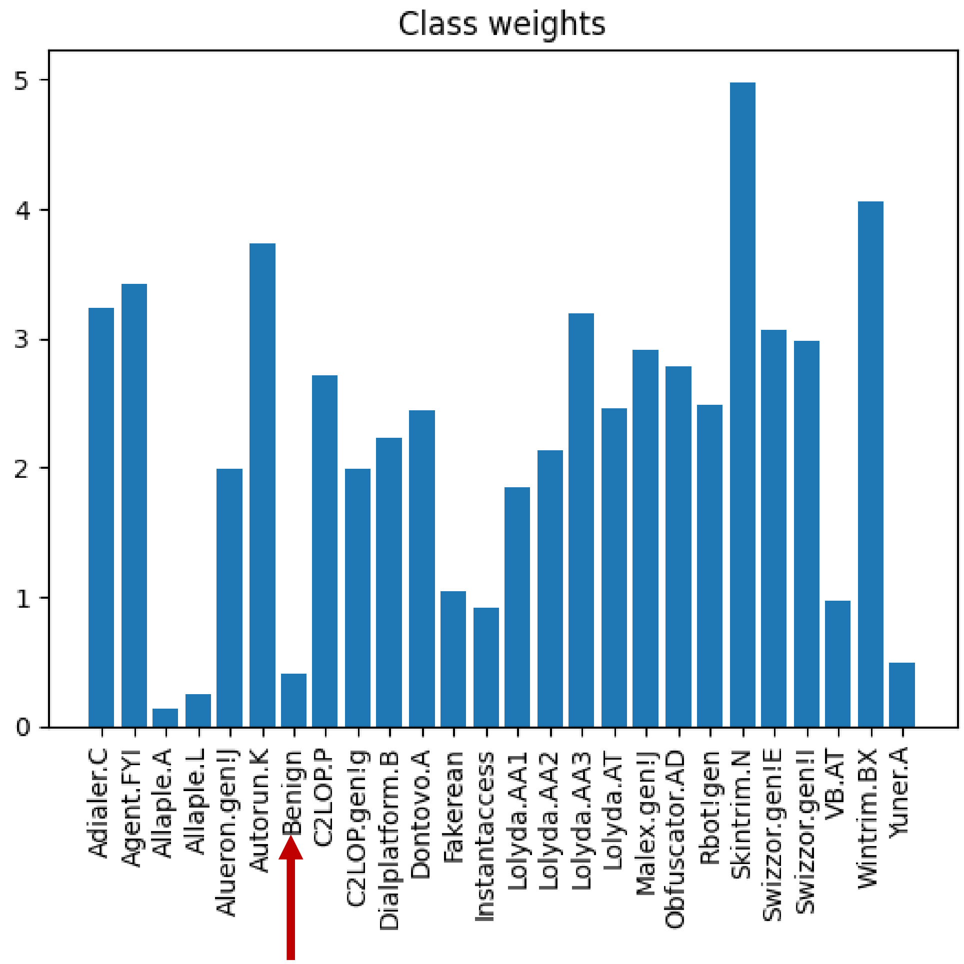






Adding a benign class to the Dataset



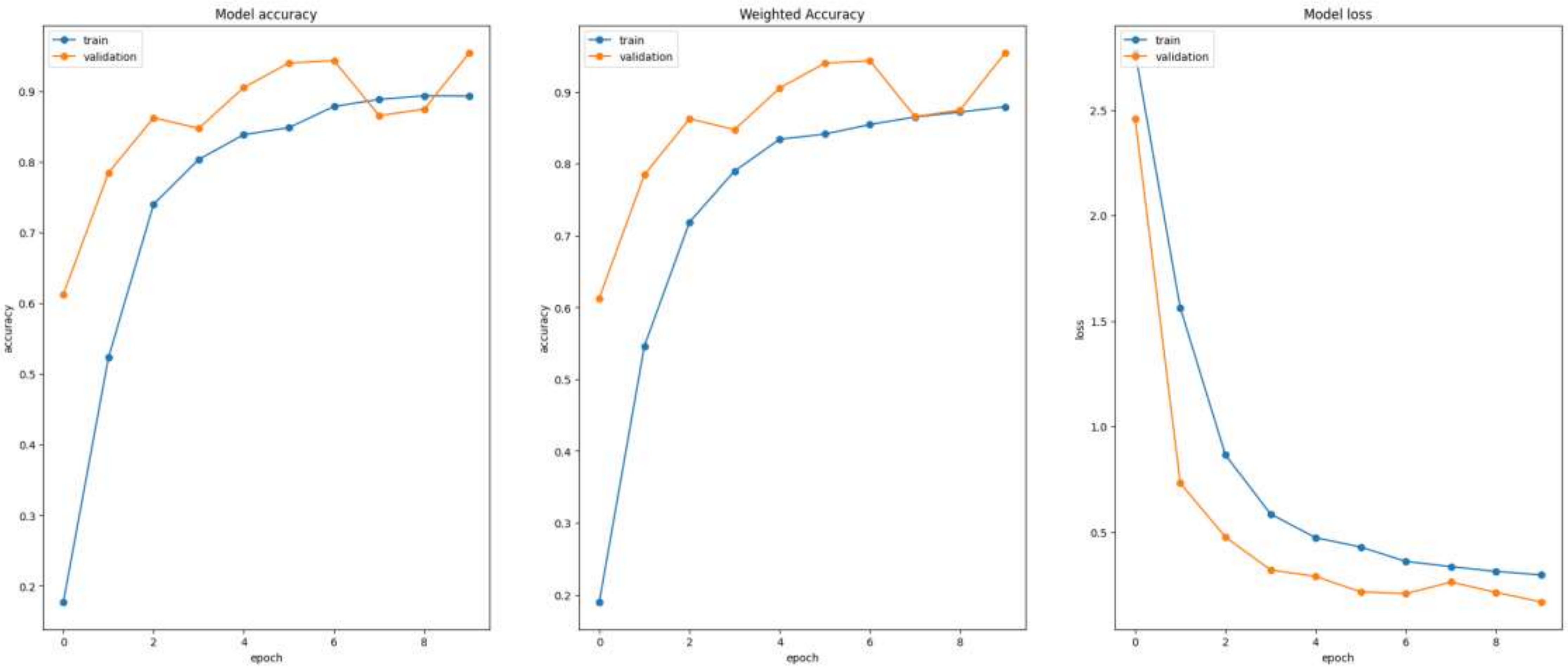


Training phase on combined dataset

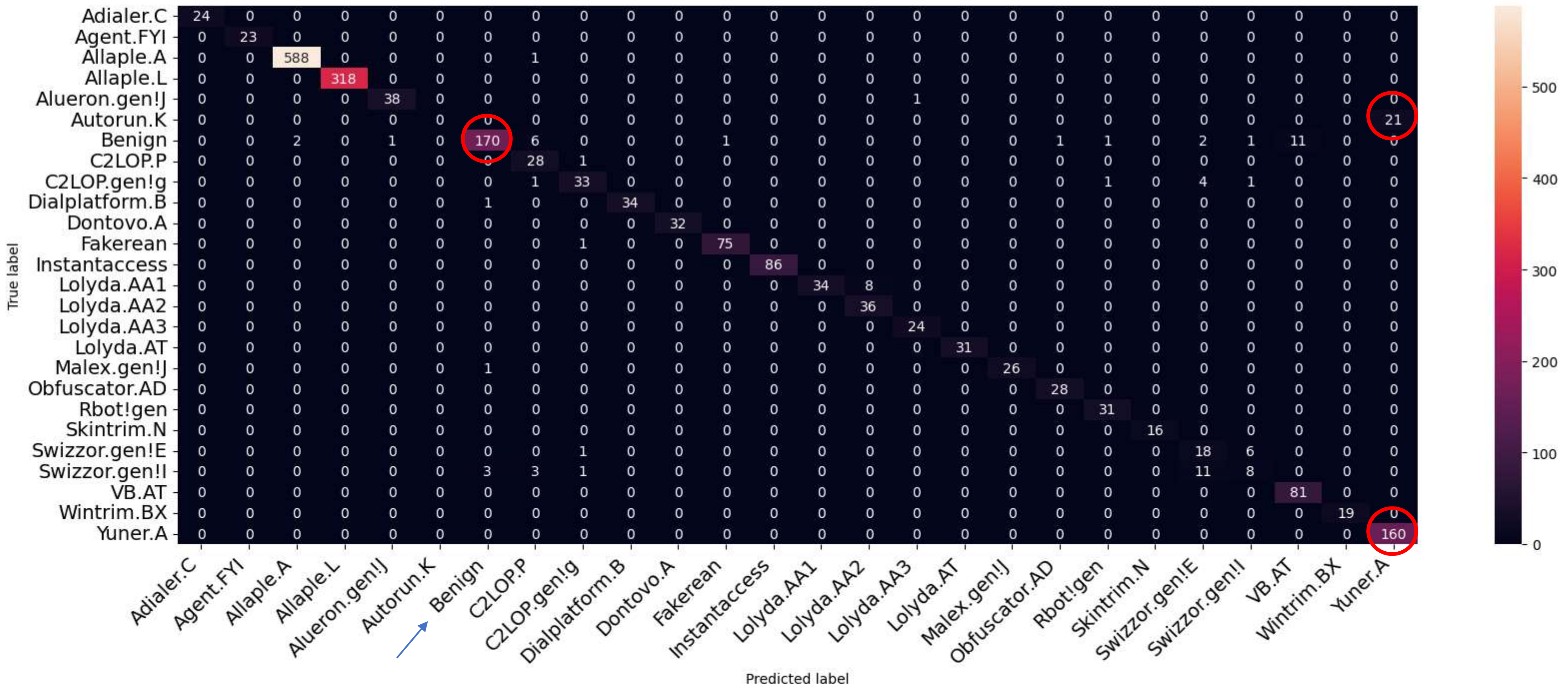
- Training on 10 epochs.
- Using the validation data.
- Using the class weights to balance the dataset.
- To evaluate the quality of training different metrics are plotted:
  - Training loss and validation loss;
  - Accuracy and validation accuracy;
  - Weighted accuracy and validation weighted accuracy;ù

- Average metrics obtained on the validation set (2054 samples):
  - loss: 0.1689 – val_accuracy: 0.9547 – val_weighted_accuracy: 0.9547

Training phase on combined dataset



Evaluation phase on validation set



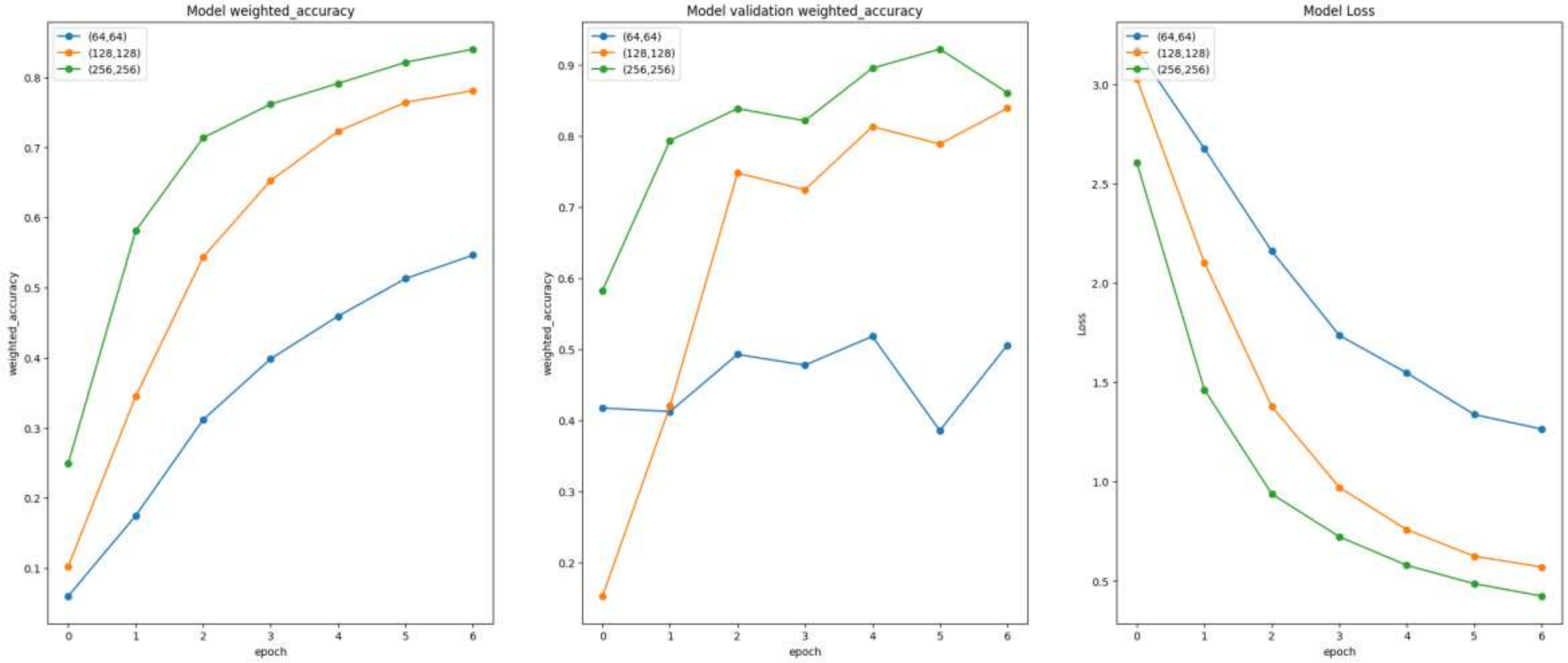
Evaluation phase on validation set

- On the malimg dataset, same misclassifications as the previous model:
  - A lot of misclassification to Yuner.A from the Autorun.K samples.
  - Swizzor.gen!E and Swizzor.gen!I.

- The model performed better on the classes by using a higher image resolution:
  - from (64,64) to (256,256).

Class	precision	recall	f1-score	support
Adialer.C	1.0	1.0	1.0	24.0
Agent.FYI	1.0	1.0	1.0	23.0
Allaple.A	0.99661	0.9983	0.99746	589.0
Allaple.L	1.0	1.0	1.0	318.0
Alueron.gen!J	0.97436	0.97436	0.97436	39.0
Autorun.K	0.0	0.0	0.0	21.0
Benign	0.97143	0.86735	0.91644	196.0
C2LOP.P	0.71795	0.96552	0.82353	29.0
C2LOP.gen!g	0.89189	0.825	0.85714	40.0
Dialplatform.B	1.0	0.97143	0.98551	35.0
Dontovo.A	1.0	1.0	1.0	32.0
Fakerean	0.98684	0.98684	0.98684	76.0
Instantaccess	1.0	1.0	1.0	86.0
Lolyda.AA1	1.0	0.80952	0.89474	42.0
Lolyda.AA2	0.81818	1.0	0.9	36.0
Lolyda.AA3	0.96	1.0	0.97959	24.0
Lolyda.AT	1.0	1.0	1.0	31.0
Malex.gen!J	1.0	0.96296	0.98113	27.0
Obfuscator.AD	0.96552	1.0	0.98246	28.0
Rbot!gen	0.93939	1.0	0.96875	31.0
Skintrim.N	1.0	1.0	1.0	16.0
Swizzor.gen!E	0.51429	0.72	0.6	25.0
Swizzor.gen!I	0.5	0.30769	0.38095	26.0
VB.AT	0.88043	1.0	0.93642	81.0
Wintrim.BX	1.0	1.0	1.0	19.0
Yuner.A	0.88398	1.0	0.93842	160.0
accuracy	0.95472	0.95472	0.95472	0.95472
macro avg	0.88465	0.89958	0.8886	2054.0
weighted avg	0.94798	0.95472	0.94947	2054.0

Effects of target image size on training

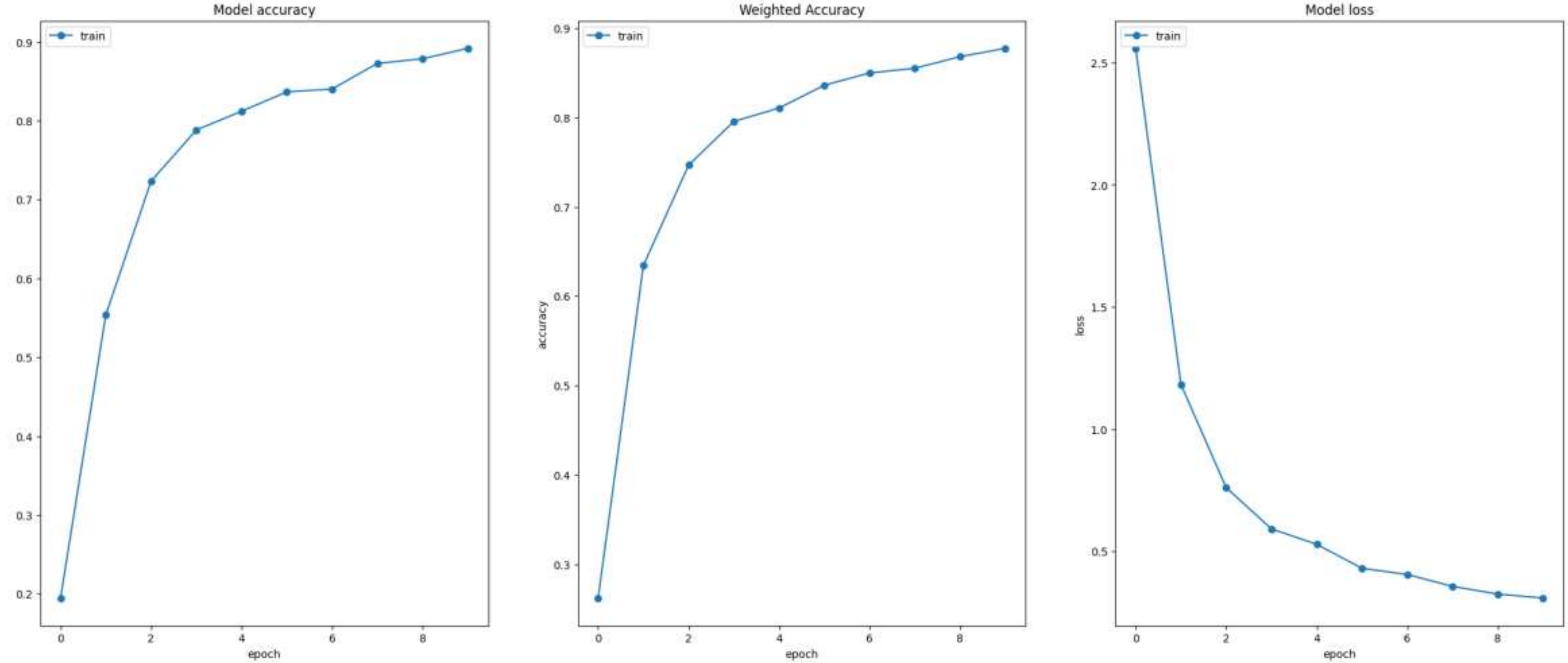


Retraining phase on combined dataset

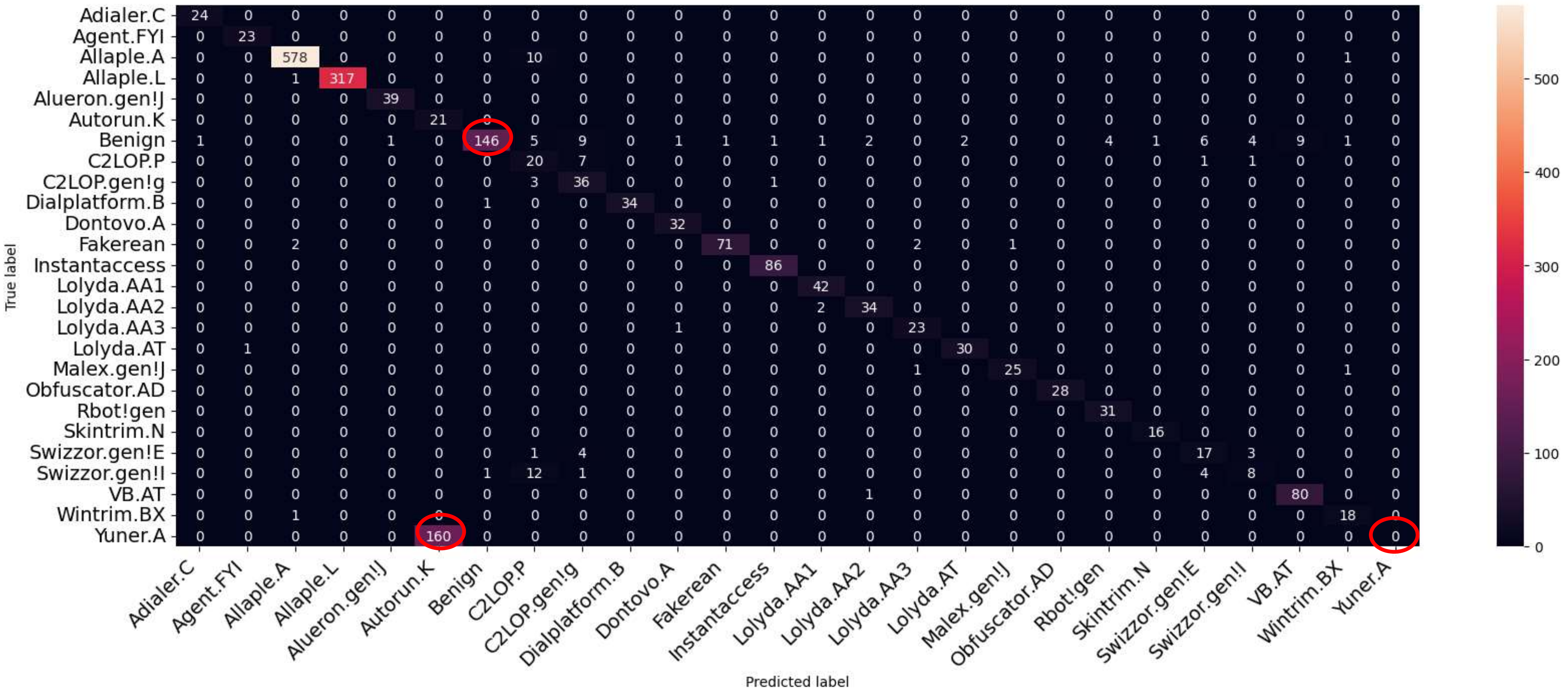
- Training on 10 epochs.
- Using the class weights to balance the dataset.
- To evaluate the quality of training different metrics are plotted:
  - Training loss and validation loss;
  - Accuracy and validation accuracy;
  - Weighted accuracy and validation weighted accuracy;

- Average metrics obtained on the test set (2053 samples):
  - loss: 0.3082 – accuracy: 0.8919 – weighted_accuracy: 0.8773

Retraining phase on combined dataset



Evaluation phase on test set

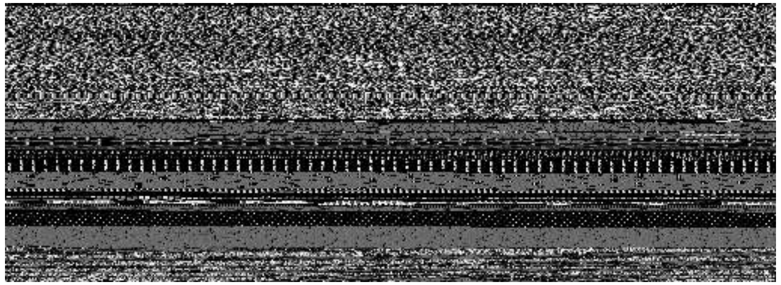


Evaluation phase on test set

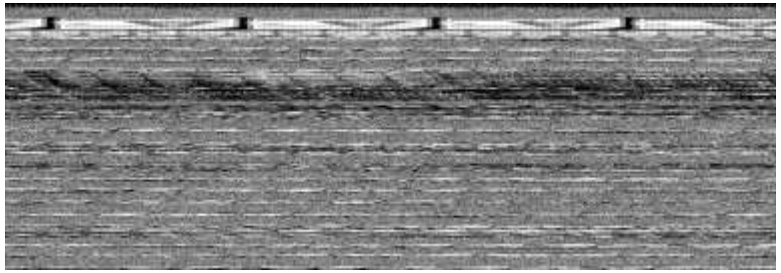
- Very similar performance to the previous training, except for Yuner.A and Autorun.K samples.
- Slightly lower performances on the benign samples than the previous training.
- Despite the presence of a benign class, other malicious classes performed well.

Class	precision	recall	f1-score	support
Adialer.C	0.96	1.0	0.97959	24.0
Agent.FYI	0.95833	1.0	0.97872	23.0
Allaple.A	0.99313	0.98132	0.98719	589.0
Allaple.L	1.0	0.99686	0.99843	318.0
Alueron.gen!J	0.975	1.0	0.98734	39.0
Autorun.K	0.11602	1.0	0.20792	21.0
Benign	0.98649	0.74872	0.85131	195.0
C2LOP.P	0.39216	0.68966	0.5	29.0
C2LOP.gen!g	0.63158	0.9	0.74227	40.0
Dialplatform.B	1.0	0.97143	0.98551	35.0
Dontovo.A	0.94118	1.0	0.9697	32.0
Fakerean	0.98611	0.93421	0.95946	76.0
Instantaccess	0.97727	1.0	0.98851	86.0
Lolyda.AA1	0.93333	1.0	0.96552	42.0
Lolyda.AA2	0.91892	0.94444	0.93151	36.0
Lolyda.AA3	0.88462	0.95833	0.92	24.0
Lolyda.AT	0.9375	0.96774	0.95238	31.0
Malex.gen!J	0.96154	0.92593	0.9434	27.0
Obfuscator.AD	1.0	1.0	1.0	28.0
Rbot!gen	0.88571	1.0	0.93939	31.0
Skintrim.N	0.94118	1.0	0.9697	16.0
Swizzor.gen!E	0.60714	0.68	0.64151	25.0
Swizzor.gen!I	0.5	0.30769	0.38095	26.0
VB.AT	0.89888	0.98765	0.94118	81.0
Wintrim.BX	0.85714	0.94737	0.9	19.0
Yuner.A	0.0	0.0	0.0	160.0
accuracy	0.86654	0.86654	0.86654	0.86654
macro avg	0.81705	0.88236	0.8316	2053.0
weighted avg	0.86601	0.86654	0.85951	2053.0

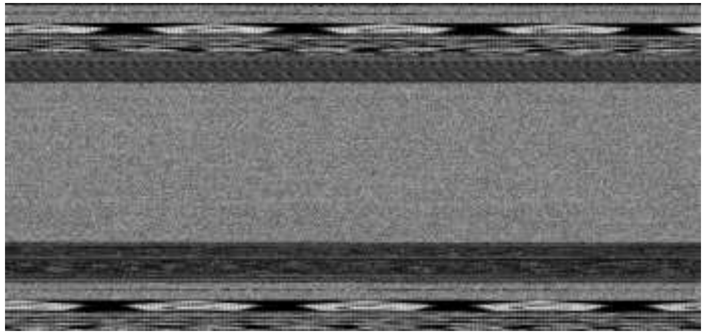
Prediction examples from the Test set



True class: Benign
Predicted class: Benign
Probability: 1.0

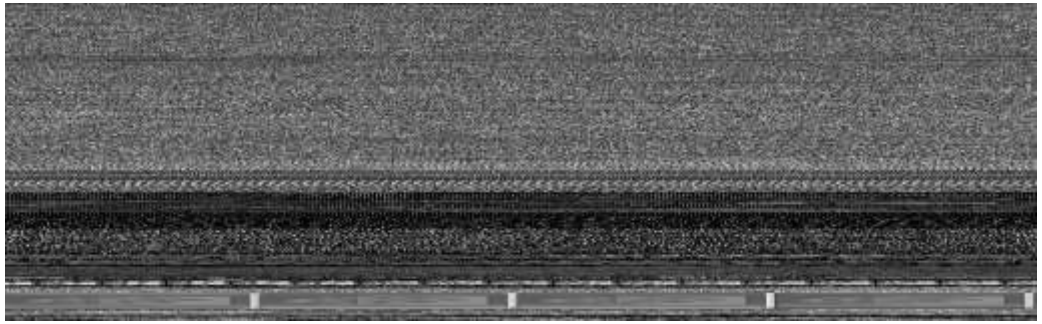


True class: Benign
Predicted class: VB.AT
Probability: 0.88

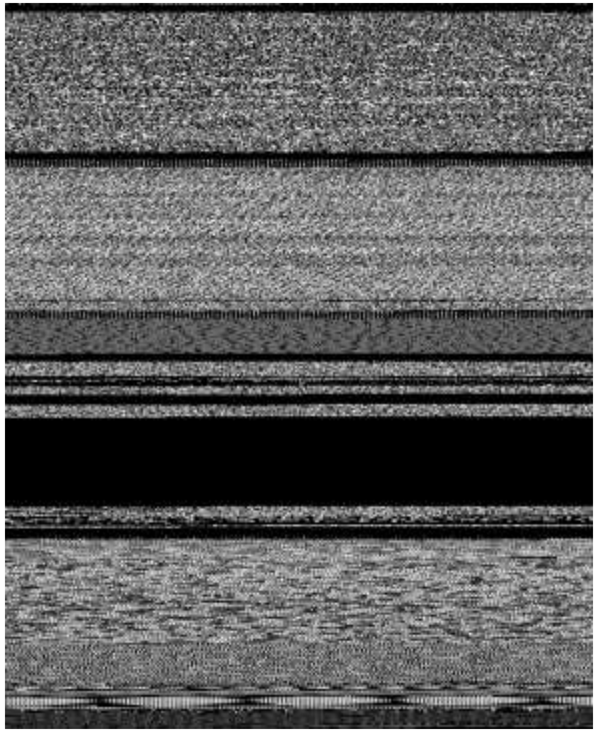


True class: Benign
Predicted class: Fakerean
Probability: 0.90

Prediction examples from the Test set

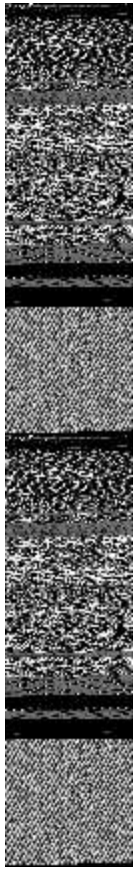


True class: Benign
Predicted class: Benign
Probability: 0.58

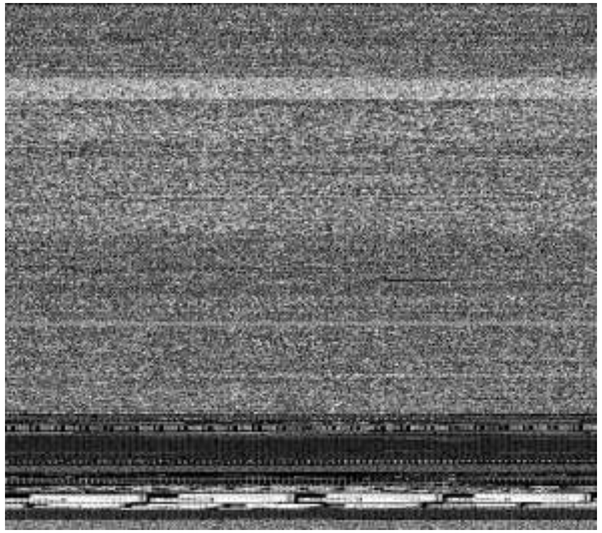


True class: Swizzor.gen!I
Predicted class: Benign
Probability: 0.89

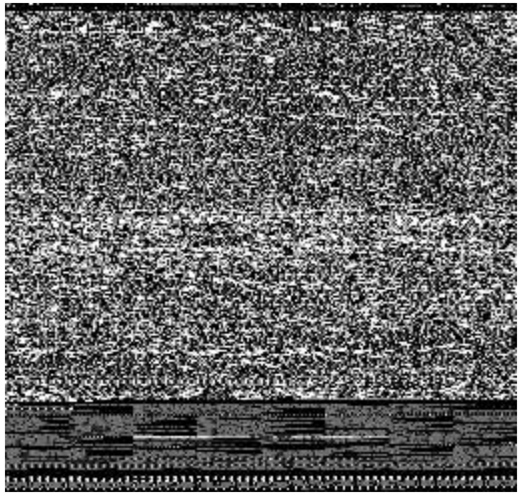
Prediction examples from the Test set



True class: Dialplatform.B
Predicted class: Benign
Probability: 0.34

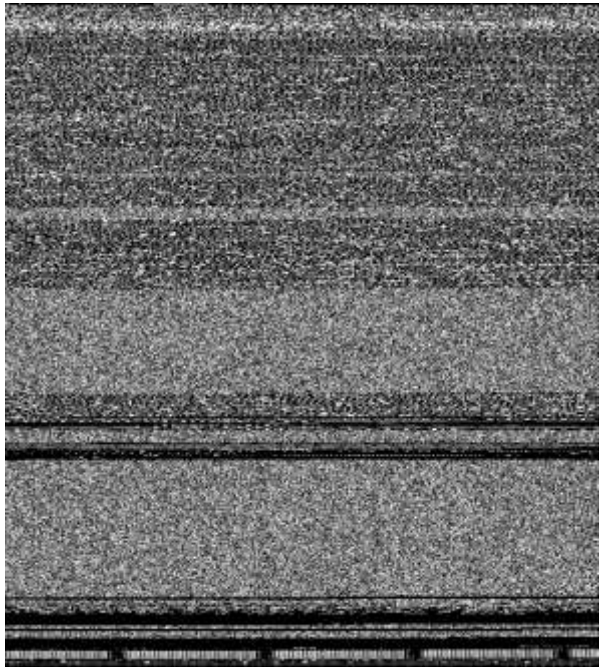


True class: Yuner.A
Predicted class: Autorun.K
Probability: 0.62

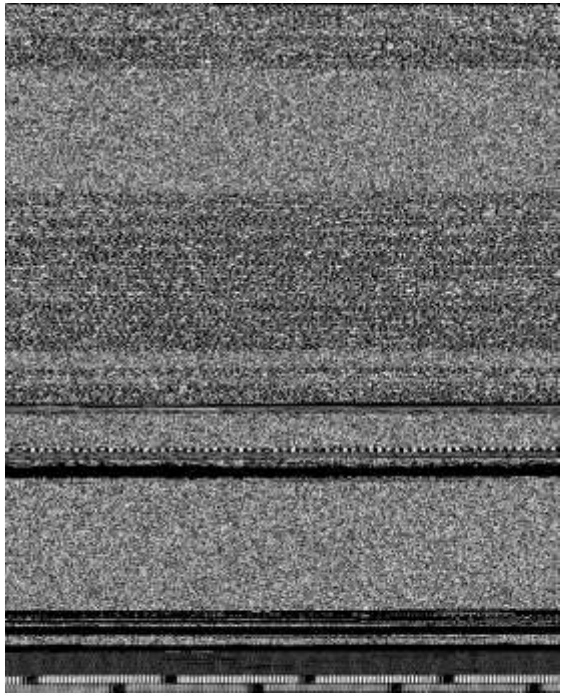


True class: Benign
Predicted class: Benign
Probability: 0.66

Prediction examples from the Test set



True class: Swizzor.gen!I
Predicted class: Swizzor.gen!E
Probability: 0.53



True class: Swizzor.gen!E
Predicted class: Swizzor.gen!I
Probability: 0.55

Conclusions

- The gray scale image representation of executables has some drawbacks related to how images are generated:
  - New hyperparameter to tune: image size.
  - Imposing spatial correlation between pixels in different rows, which is not always true.
  - As seen in the experiments, the approach suffers from code obfuscation and encryption (see Yuner.A and Autorun.K), which might completely change the bytes structure.

- Although the drawbacks, the final model can differentiate between malicious and benign data.

- The malware detection task can be improved by:
  - Aggregating all the malware classes under one malicious class;
  - Collecting more benign samples in the wild;

Future work

- A more generalizable approach is the multimodal learning where different feature vectors, belonging from different inputs of the PE executable (strings, api calls, control flow graphs ecc.), can be used.
  - For each feature vector there is a classifier;
  - A fusion layer gathers all the predictions to decide the final output.

- Because of the continuos evolution of malware and its variants, another important task to achieve is the class incremental learning:
  - a model, pretrained on a set of malware classes, gains new knowledge by learning new malware classes without forgetting the old ones.

References

Gibert, D., Mateu, C., Planes, J. *et al.* Using convolutional neural networks for classification of malware represented as images. Using convolutional neural networks for classification of ... – Springer.

Daniel Gibert, Carles Mateu, Jordi Planes, Journal of Network and Computer Applications, The rise of machine learning for detection and classification of malware: Research developments, trends and challenges. The rise of machine learning for detection and ... – ScienceDirect.

Songqing Yue, Tianyang Wang, Imbalanced Malware Images Classification: a CNN based Approach. Imbalanced Malware Images Classification: a CNN based Approach.

Nataraj, Lakshmanan & Karthikeyan, Shanmugavadivel & Jacob, Grégoire & Manjunath, B.. (2011). Malware Images: Visualization and Automatic Classification. 10.1145/2016904.2016908. Malware Images: Visualization and Automatic Classification – ResearchGate.

M. Kalash, M. Rochan, N. Mohammed, N. D. B. Bruce, Y. Wang and F. Iqbal, "Malware Classification with Deep Convolutional Neural Networks," 2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS), Paris, France, 2018, pp. 1-5, doi: 10.1109/NTMS.2018.8328749. Malware Classification with Deep Convolutional Neural Networks | IEEE ...

Tuan, Anh Pham; Phuong, An Tran Hung; Thanh, Nguyen Vu; Van, Toan Nguyen (2018). Malware Detection PE-Based Analysis Using Deep Learning Algorithm Dataset. figshare. Dataset. Malware Detection PE-Based Analysis Using Deep Learning Algorithm Dataset