

École polytechnique de Louvain

Let's visualize malwares!

Malware visualization and classification

Author: **Benoît MICHEL**

Supervisor: **Axel LEGAY**

Readers: **Charles-Henry BERTRAND VAN OUYTSEL, Christophe DE VLEESCHOUWER, Fabien DUCHÊNE, Kim MENS**

Academic year 2020–2021

Master [120] in Computer Science and Engineering

Acknowledgments

This master thesis and the work behind it would not have been possible without the support of my supervisor, Axel Legay. His motivation, ideas and support allowed me to believe in myself and kept my work on track from our first encounter to the final delivery. I would also like to thank Fabien Duchêne and Charles-Henry Bertrand Van Ouytsel, for their continuous supervision throughout the semester, their frankness and their good ideas. I am glad to have had such a nice supervision team.

Besides, nobody gave me more ideas during this thesis than my roommate, Pierre-François De Plaen. His knowledge of machine learning, availability and humor to decompress were very useful to me.

Finally, nothing would be possible without my family. I would like to thank my parents, whose love and guidance are with me in whatever I pursue and I would like to thank my numerous siblings for their proofreading, their comments and their unending inspiration.

Abstract

Due to the growing threat from malicious softwares, security of sensitive networks is becoming increasingly crucial. The ever-growing number of suspicious samples is a time-consuming process for malware analysts and the need is real for a tool able to quickly analyze a large number of samples. The combination of new visualization methods from dense pixel displays based on raw sample data, combined with promising technologies for the image classification such as convolutional neural networks, is able to meet this need. In this master thesis, we provide two new visualization techniques and a classifier able to categorize new samples into malware families at the entry of a high-speed network with an accuracy up to 97.82%. In case of doubt, samples can still be analyzed in more depth by malware experts. Additionally, we explore different datasets, the impact of their structure on classification accuracy and we analyze the insertion of cleanwares in the dataset.

The whole implementation of our work is freely accessible on GitHub:
https://github.com/Tioneb88/TFE_Malware_Visualization_and_Classification

Contents

List of Abbreviations	vii
List of Figures	viii
List of Tables	xii
1 Introduction	1
2 Background	3
2.1 Visualization generation	3
2.1.1 Data extraction	4
2.1.2 Visualization techniques	5
2.2 Machine learning and classification	9
2.2.1 Artificial neural networks	9
2.2.2 Multi layer perceptrons	12
2.2.3 Convolutional neural networks	15
2.3 Summary	19
3 Visualizations	20
3.1 Extraction method	20
3.2 Visualization method	21
3.3 Generation of the visualizations	22
3.3.1 Mapping	22
3.3.2 Size of the visualizations	22
3.3.3 Used visualizations	23
3.3.4 RGB tuned: channels decomposition	26
3.3.5 Embedded images	27
3.3.6 Study of family diversity	28
4 Classification	31
4.1 Our architectures	31
4.1.1 MLP	31

4.1.2	CNN	32
4.2	Dataset apportionment	33
4.3	Choice of fixed parameters	33
4.3.1	Epochs	34
4.3.2	Input shape	34
4.3.3	Activation function	35
4.3.4	Padding	36
4.3.5	Metrics	36
4.3.6	Loss function	37
4.3.7	Optimizer and learning rate	37
5	Experiments and Results	40
5.1	Presentation of the datasets	40
5.1.1	Cisco dataset	40
5.1.2	Microsoft malware classification challenge on Kaggle	41
5.1.3	Cleanwares	41
5.1.4	Custom dataset	42
5.2	Implementation	43
5.2.1	Hardware configuration	43
5.2.2	Python implementation	43
5.3	Generation time of visualizations	44
5.4	Classification experiments	44
5.4.1	Comparison of classifiers	45
5.4.2	Comparison of visualizations accuracy	46
5.4.3	Comparison with Kaggle	47
5.4.4	Comparison with literature	48
5.4.5	Analysis and dissection of Cisco dataset	49
5.4.6	Classification with cleanwares	50
6	Conclusion and further work	52
	Bibliography	54
A	Malware visualizations by family	58
B	Presentation of the malware families	64
B.1	Cisco dataset	64
B.2	Kaggle dataset	66
B.3	Other datasets	67

List of Abbreviations

AI	Artificial Intelligence
ANN	Artificial Neural Network
API	Application Programming Interface
BP	BackPropagation
CCE	Categorical Cross-Entropy
CNN	Convolutional Neural Network
DLL	Dynamic Link Library
MBP	Markov Byte Plot
ML	Machine Learning
MLP	MultiLayer Perceptron
Nmap	Neighborhood map
PE	Portable Executable
PE	Python Imaging Library
ReLU	Rectified Linear Unit
RGB	Red Green Blue
SHA	Secure Hash Algorithm

List of Figures

2.1	3D scatter plot: number of calls to some functions of some libraries over time [13]	6
2.2	Bar chart: number of permissions features occurrences for benign (light blue) and malware (dark blue) classes [4]	6
2.3	Scatter plot matrice: comparison of cars under different characteristics [14]	7
2.4	Stardimates: each direction is a characteristic of the sample [15] . .	7
2.5	Chernoff faces: each element of the face is linked to a feature and changes according to its value [16]	7
2.6	Needle icons: calls between states, density represented by color, angle and depth [12]	7
2.7	Treemap: used system call operations for an individual malware sample [4]	8
2.8	Dimensional stacking: projection of high-dimensional data by embedding dimensions within other dimensions [17]	8
2.9	Matrix visualization: similar malware samples are arranged next to each other and assigned similar color values to visualize commonalities [18]	9
2.10	Circle segment technique: classification of Wine data against 13 features [19]	9
2.11	Global structure of an ANN with deep learning: an input layer, one or more hidden layers and an output layer; edges connectivity and the number of nodes may vary	11
2.12	Most used activation functions on one fold from the previous layers	13
2.13	Typical CNN architecture [30]	15
2.14	Convolution operation [32]	16
2.15	Max pooling with filter 2×2 and stride = 2 [34]	17
2.16	Illustration of a neural network before (left) and after (right) the application of a dropout, inspired from [35]	18
3.1	The hexadecimal representation of a sample (in the red rectangle) [36]	22
3.2	A zoom to the pixels of a picture [37]	22

3.3	Grayscale visualizations from 3 different malware families: 1click-download (SHA256: 00a23ccd 9a348812 00c269bf dbf9b293), ainslot (SHA256: 00e947cc 0d0cc9c0 f46dbf9d d5cd33b9) and allaple (SHA256: 00023154 6b0e5eb4 306b31c6 81b4d366) (from left to right)	24
3.4	RGB visualizations from three different malware families: 1click-download (SHA256: 00a23ccd 9a348812 00c269bf dbf9b293), ainslot (SHA256: 00e947cc 0d0cc9c0 f46dbf9d d5cd33b9) and allaple (SHA256: 00023154 6b0e5eb4 306b31c6 81b4d366) (from left to right); some visualizations may appear distorted due to the layout	24
3.5	RGB tuned visualizations from three different malware families: 1clickdownload (SHA256: 00a23ccd 9a348812 00c269bf dbf9b293), ainslot (SHA256: 00e947cc 0d0cc9c0 f46dbf9d d5cd33b9) and allaple (SHA256: 00023154 6b0e5eb4 306b31c6 81b4d366) (from left to right) (from left to right)	25
3.6	RGB tuned visualization from the amonetize (SHA256: 00faa8a8 7f86db20 ab8c9011 3889118e) malware family	26
3.7	RGB tuned visualization from the amonetize (SHA256: 00faa8a8 7f86db20 ab8c9011 3889118e) malware family decomposed by color channel	27
3.8	RGB tuned visualization from the autoit (SHA256: 011c7cb1 7d6d0af5 604a8fc2 c2bb34f5) malware family with an embedded image	28
3.9	Similar visualizations of three malwares from the blackmoon family (from left to right, SHA256: 002f34b3 60261bda 7f691a3a 96aa83e0, SHA256: 004b731e bfce328e 10c0bfe8 7e9fa872 and SHA256: 004c0f4f 6d6cbd75 31fba8a e9834fe2)	29
3.10	Different visualizations of three malwares from the blackmoon family (from left to right, SHA256: 005c24d4 fb89d9c1 3677f203 55f3fe4a, SHA256: 00a9263f 2a056e62 631e0437 b80dd482 and SHA256: 0075c094 f5cbe6f1 f4949b33 09508558)	30
4.1	Variation of precision in percentage at the entry of MLP and CNN according to input shape	34
4.2	Variation of time in seconds at the entry of MLP and CNN according to input shape	34
4.3	Variation of precision in percentage of MLP and CNN according to activation function	35
4.4	Variation of time in seconds of MLP and CNN according to activation function	35
4.5	Variation of precision in percentage of MLP and CNN according to final activation function	35

4.6	Variation of time in seconds of MLP and CNN according to final activation function	35
4.7	Variation of precision in percentage of MLP and CNN according to loss function	37
4.8	Variation of time in seconds of MLP and CNN according to loss function	38
4.9	Variation of precision in percentage for MLP and CNN according to optimizer	39
4.10	Variation of time in seconds for MLP and CNN according to optimizer	39
5.1	Percentage of malwares by family in the Cisco dataset	41
5.2	Percentage of malwares by family in the Kaggle dataset	42
5.3	Percentage of malwares by family in the custom dataset	43
5.4	Leaderboard of the Microsoft malware classification challenge on Kaggle [46]	47
5.5	Confusion matrix of the classification with CNN on Kaggle dataset	48
5.6	Percentage of accuracy of the CNN on the Cisco dataset with variable number of families	49
5.7	Number of malwares in the Cisco dataset with variable number of families	50
5.8	Mean number of malwares by family in the Cisco dataset with variable number of families	50
A.1	Families benjamin (SHA256: 000c0b57 42e212bf 10c27e5f ea3f9e84), ausiv (SHA256: 000cc189 b87f2bb6 681ae769 66df65dc) and banload (SHA256: 000cde2d 5d5ba78f 189a3151 5d0e945e) (from left to right)	58
A.2	Families browsefox (SHA256: 00a2a587 7bbc621f 1e8d7eaf 1f24e68a), drolnux (SHA256: 000af4fa 5083ad36 aaa2fd0a 52381954) and lamer (SHA256: 000c5408 cfc66c69 7285d1bf 89c87650) (from left to right)	59
A.3	Families qbot (SHA256: 00c6e021 e85df725 00baef46 590c993e), rozena (SHA256: 0073e91a 7b9d95dd 759b8fea 5dbe57bb) and shipup (SHA256: 000abb36 c450fec4 b89ea87d b7b91b14) (from left to right)	59
A.4	Families gobot (SHA256: 011e978c 685d5a12 bda6c98c 73a2dbf3), high (SHA256: 0000b21d d4ac576d dba95156 7f2edf00) and install-core (SHA256: 000c6a48 e3bbd31a 23df66c3 9ddc854d) (from left to right)	60
A.5	Families mansabo (SHA256: 00a131e4 742b6d99 23e838e5 4cb00e22), nitol (SHA256: 000c95eb d4647282 a2cba525 6be4c1b0) and pioneer (SHA256: 00f2b2bf f0e04078 6d6ff394 57770bf9) (from left to right)	60

A.6	Families bancteian (SHA256: 000acd72 ac2628c8 733533dc a4afbb30), ceeinject (SHA256: 000c1c50 218d6a0e 5114670c 5e253273) and installmonster (SHA256: 000eab0e c8cb006d 6eb3a8c1 0b8a9b01) (from left to right)	61
A.7	Families downloadguide (SHA256: 0000f512 4830ef17 ee2091ab 45c502e3), delf (SHA256: 00a23cee b488befd fb90ad4e e42aba84) and wapomi (SHA256: 00b69bea 8f6611a4 1395932f 342bba9d) (from left to right)	62
A.8	Families cleanwares (SHA512: 00ad45fb 1aa57b6e 6f09be50 3ee2f355 78bbff87 b5da2bf3 87553422 e8f310e9), vtflooder (SHA256: 012a2fef 46cfa4df 64007ab9 932f7e6e) and gandcrab (SHA256: 00a7c8f3 f21a3d5f 769653e7 d922dcd7) (from left to right)	62
A.9	Families stihat (SHA256: 000ca68a dacfdb1a 92eb533c e6c2f132), upatre (SHA256: 000c082b c55e68e4 51fd6ce3 a435b573) and ipamor (SHA256: 00b1b058 696b1afe 86405925 9ad2c9e7) (from left to right)	63
A.10	Families wabot (SHA256: 000cecdde 8032aff7 93e4de4f a42b41a6), dinwod (SHA256: 000ab4c9 a3f90e3f 8528857a 5c7dfb35) and zusy (SHA256: 000a8e50 c83cf1ad 32539e0e c37d05c0) (from left to right)	63

List of Tables

3.1	Fixed sizes of the visualizations according to the size of the samples	23
5.1	Time in seconds for the generation of visualizations on several datasets: above grayscale and RGB, and below RGB tuned and disassembled	44
5.2	Comparison of accuracy in percentage between our MLP and our CNN on the Cisco dataset, for variable parameters of the dense and dropout layers, using RGB tuned visualization method	45
5.3	Comparison of accuracy in percentage for our three visualization methods on the custom dataset, for variable parameters of the dense and dropout layers, using our CNN	46
B.1	Distribution of the malwares by family in the Cisco dataset	66
B.2	Distribution of the malwares by family in the Kaggle dataset	67

Chapter 1

Introduction

Nowadays the use of software is constantly growing and publishers must always create more to satisfy a hungry market. In this prosperous industry, reliability and security are too often relegated to the background. Hackers have understood this so that more and more of them are trying to bypass systems. Malicious software, or malware, can be defined as "any software that does something that causes harm to a user, computer, or network" [1]. A malware can be a virus able to hide into a software and run when his host is requested, a worm slipping into communication tools and copy himself wherever he can, a Trojan horse where the malicious code is disguised behind a cleanware, i.e. a friendly software, a backdoor able to keep continually open a door for a hacker on a machine and so on. Malwares can have devastating effects on privacy, economy, medical system, energy supply management, defense, mobility, information or even national interests [2].

In 2019, over 439 000 new malware variants (12.3% increase compared to 2018) were detected and in the same time, 62% of cybersecurity professionals believed their organization's cybersecurity team was understaffed [3]. To counter this lack of security professionals, the use of rapid and accurate automated detection methods is more necessary than ever. The goal of this work is to develop a tool able to quickly analyze and classify a high-speed flow of new samples, for example to secure the input data stream of a network. This tool would be fast enough to not slow down the input rate of the network but precise enough to make good classification and spare the cybersecurity experts precious time. Only in case of doubt, the sample could be dissected with a human deeper and slower look.

In recent years, visualization-based approaches [4] have been proposed to directly analyze malware binaries, i.e. the raw malware data, for classification, which further improves the efficiency of malware detection because no in-depth analysis is required. The speed of the evaluation is determined and this approach is well adapted to a constant flow of new inputs. The idea of this work is to combine the latest

visualization techniques with the power of highly accurate Convolutional Neural Networks (CNNs).

Since the AlexNet CNN in 2012, deep learning algorithms and CNNs are in the lead for images classification [5]. To quantify the interest of using a CNN, we will compare its accuracy with a MultiLayer Perceptron (MLP), another supervised learning classifier.

Other papers [6, 7, 8] have already been interested in the subject, proposing new visualization approaches with RGB-colored images and disassembled code, using CNN as classifier and studying adversarial examples, i.e. carefully crafted inputs created by adding a small perturbation to the original input and producing the adversaries' desirable outputs, such as misclassification. But we want to introduce new, faster and more accurate visualization methods with no disassembling or execution step. We also want to analyze the classifiers behavior with different datasets and with a mix of malwares and cleanwares. Furthermore, our approach can be generalized to other fields than malware classification and cybersecurity, as soon as a visualization is possible.

With our new visualization methods and our classifiers, we get classification results up to 97.82% accuracy with a less time greedy generation. We have carried out experiments on various datasets and we also explored the structure of datasets that can explain our classification results and how to improve them.

The content of this document is organized as follows. The underlying theory of this work and all you need to read and understand the following chapters is detailed in the chapter 2 relative to the background. The extraction of the data and the different visualization techniques we used are presented in the chapter 3. Chapter 4 provides an overview of the layers and the parameters used for our MLP and CNN. Then, chapter 5 describes the datasets we used, the experiments that we realized, our results and discussions related to the literature. Finally, the limitations of our approach, conclusion and further work are discussed in chapter 6.

Chapter 2

Background

In this chapter, we will investigate all the visualization techniques that we could use to visualize a malware. We search for a method able to quickly generate an image while preserving a maximum of information to ensure good precision in the classification process. We will also explore different classifiers using deep learning. In this work, we want to create a tool fast enough to be able to process a continuous flow of new samples and precise enough to be able to classify them, using a CNN. The choice of a CNN as classifier will be justified in the chapter 4. Before explaining our solution, let us find out and understand what already exists.

2.1 Visualization generation

There is no consensus in the community as to which of the representations is generally better suited for visualization [4, 9]. There exists a lot of different techniques, each with their benefits and drawbacks. We will need to consider different options to choose the more suitable method for our work.

Let us find a method capable of extracting as much information as possible from a sample and transforming it into a visualization as quickly as possible. Our priorities are therefore in order: precision, speed and storage space. The precision, i.e. the proportion of well classified malwares, is highly important, for example, if we want to use this system to classify the malwares at the entry of a network, a misclassification would be disastrous since the network would be infected. The speed is also important since we want to be able to predict a family to samples as they are coming, and the input rate of a network can be high. Finally, it is also good to take into account the storage space since we need to have compact visualizations if we want to train our CNN on a big dataset.

To visualize a malware, or more generally an executable, we first need to identify the data that we will use for the representation. We can use the raw file, the

machine code, or outright run the code in a secure environment to get information about its interactions during the execution. There exists different techniques of data extraction detailed in the following section 2.1.1. Afterwards, we need to choose the adequate visualization technique for our data and our use, the most precise, fast and compact representation. We can make it in 2D or in 3D, unique or multiple, human-readable or not, interactive or not. The different categories and techniques are introduced in the section 2.1.2.

2.1.1 Data extraction

Computer scientists use two main techniques to gather information about a malicious piece of software: static analysis [7] and dynamic analysis [10].

Static analysis stands for all the techniques that do not require the sample to be executed during the analysis. The file can be partially disassembled to check the basic properties (e.g. file type, checksum), or to explore the sections and their extractable information (e.g., strings, DLL import information). To go to higher precision, the file can be rather fully disassembled and more information can be extracted (e.g. opcode, API sequences, function call graph) but it is a slow and complex operation. Regardless of the level of disassembly used, the execution environment for static analysis plays a negligible role. The static analysis can quickly capture syntax and semantic information but this approach is easily disturbed by encryption technology and code obfuscation (e.g., make a code appear different but keeping the same execution effect).

For example, *Yara*¹ is a static tool. In *Yara*, we write rules consisting of a set of strings and a boolean expression which determine its logic. If the sample matches the rule, a classification rule is applied. Here is an example of *Yara* file:

```
rule check_malware : malware
{
    meta:
        description = "Is this a malware?"

    strings:
        $a = {6A 40 68 00 42 00 00 6A 14 8D 91}
        $b = {42 4D B0 2B C1 83 C0 27 99 6A 4E 59 F7 F9}
        $c = "MALWARELOSTEVIL"

    condition:
        $a or $b or $c
}
```

¹<https://github.com/virustotal/yara>

Here, we write the rule named *check_malware*. The meta is ignored by the computer, this is just rule information for the human. We have three strings: *a*, *b* and *c*. The first two strings define a byte sequence and the last one defines a text string. The condition says that if at least one of these strings appears in the sample, the file is classified as malware.

Dynamic analysis stands for all the techniques executing the file on a host system. During the execution, a tool logs relevant information into an execution trace that can be analyzed later. This approach is more appropriate to analyze the behavioral information of a program such as system calls, network activity or registry operations. For this dynamic approach, the execution environment plays an essential role, it needs to be more robust, have sufficient resources and the monitoring tools must be undetectable for the malware since some use sandbox detection and evasion techniques to tamper the results of the analysis. So, the dynamic analysis can be more precise but the cost of time and resources for malware execution is expensive.

For example, *Cuckoo*² is a free sandbox. A sandbox is a closed and isolated environment in which we can securely execute malwares to observe their behavior. The principle of Cuckoo is simple, it uses a *VirtualBox*³ virtual machine to execute the malware, then it records the malware activity, stores all the information on the host and finally shuts down the virtual machine by restoring a snapshot to get the machine back clean and up for a new analysis.

The static and the dynamic approaches are the two main analysis families but there exists subfamilies. One of the static subfamilies was introduced by Jianwen Fu et al. [6]. The sample binary is directly used as raw information for the visualization [11]. It is very efficient because we do not need to disassemble the malware neither to execute it, we need no preprocessing step to access to the data. It is much faster while ignoring the eventual encryption of the file and the possible detection and evasion techniques but we can lose in precision since we only have access to the raw data.

2.1.2 Visualization techniques

A large number of information visualization techniques have been developed in the last decades. A good classification of these techniques, widely used as reference in the literature [4, 10, 12], has been suggested by Daniel A Keim et al. [9]. According to this model, there are five visualization families:

²<https://github.com/cuckoosandbox>

³<https://www.virtualbox.org/>

- **Standard 2D/3D displays** include all the basic visualization techniques with x-y and x-y-z plots such as scatter plots, bar charts, and line graphs (see figures 2.1 and 2.2). These techniques are suitable to gather information about a defined set of features or properties but in our case, each malware could have different features and we can not enumerate them all. To limit ourselves to a fixed set of features of interest would make us lose a lot of precision.

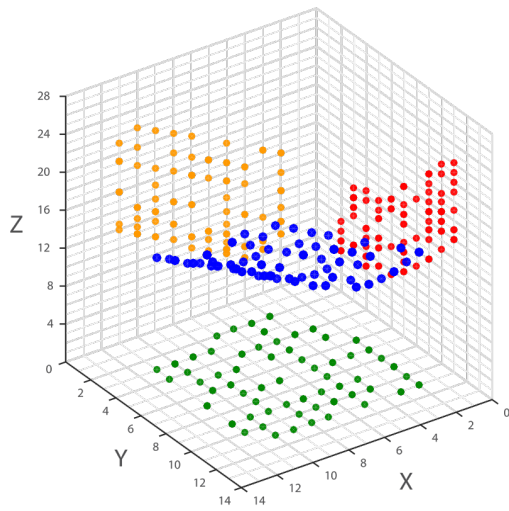


Figure 2.1: 3D scatter plot: number of calls to some functions of some libraries over time [13]

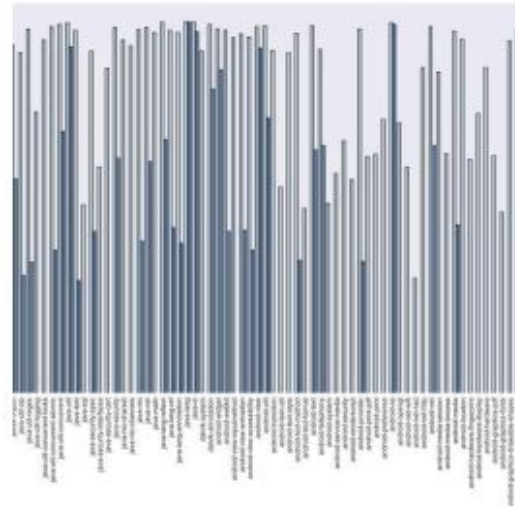


Figure 2.2: Bar chart: number of permissions features occurrences for benign (light blue) and malware (dark blue) classes [4]

- **Geometrically transformed displays** aims to visualize interesting transformations of multidimensional datasets, i.e. datasets with more than two features of interest. It includes visualizations like scatter plot matrices, node-link diagrams, parallel coordinates, and stardiates (see figures 2.3 and 2.4). These techniques are interesting to visually compare some samples but they take too much time to compute a human-readable image and once again, we need to define a set of features or properties of interest to compare, which is not appropriate in our case, for precision.
- **Icon-based displays** take attributes of multidimensional data items and map them onto the features of an icon for the representation. Some of these more creative techniques are Chernoff faces, needle icons, star icons, stick figure icons, color icons, or tile bars (see figures 2.5 and 2.6). This family of

visualization is more human-readable and artistic but again, not well adapted for a computer vision and a fast computing time. The features are too limited and need to be predefined.

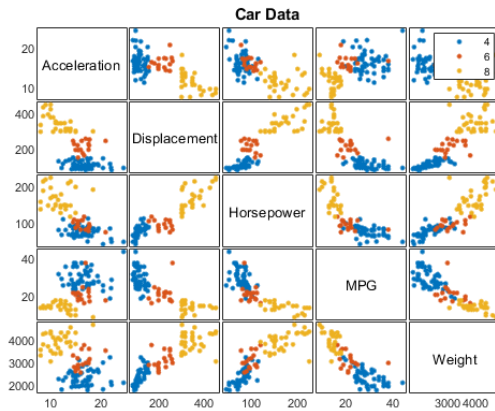


Figure 2.3: Scatter plot matrix: comparison of cars under different characteristics [14]

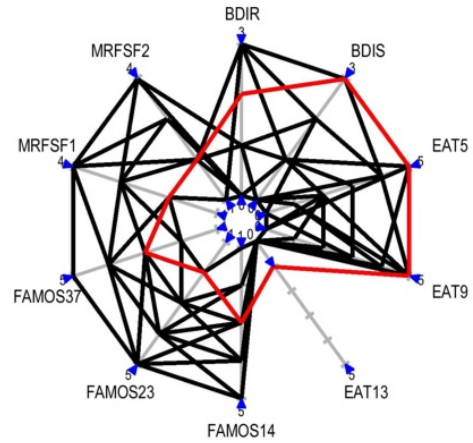


Figure 2.4: Stardimates: each direction is a characteristic of the sample [15]

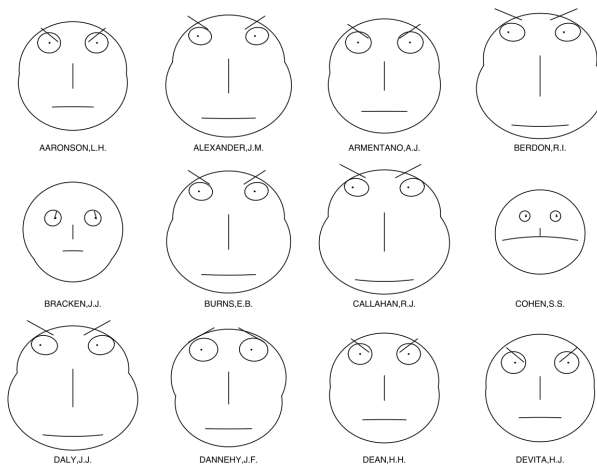


Figure 2.5: Chernoff faces: each element of the face is linked to a feature and changes according to its value [16]

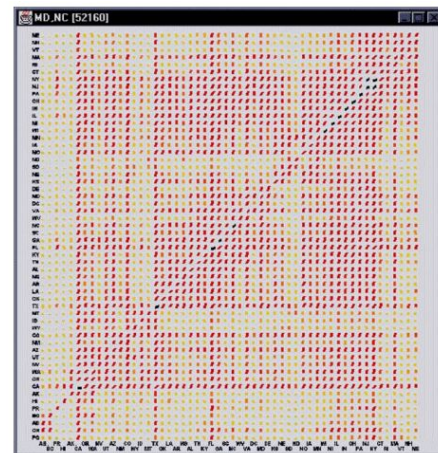


Figure 2.6: Needle icons: calls between states, density represented by color, angle and depth [12]

- **Stacked displays** are representations for hierarchical data and hierarchical layouts for multidimensional data. It includes techniques such as hierarchical stacking, treemaps, neighborhood treemaps (Nmaps), or dimensional stacking

(see figures 2.7 and 2.8). This family of visualization is appropriate for a disassembled sample but it takes time to proceed, with a more speed oriented approach we have no time to disassemble and no access to the hierarchy, making this family unusable.

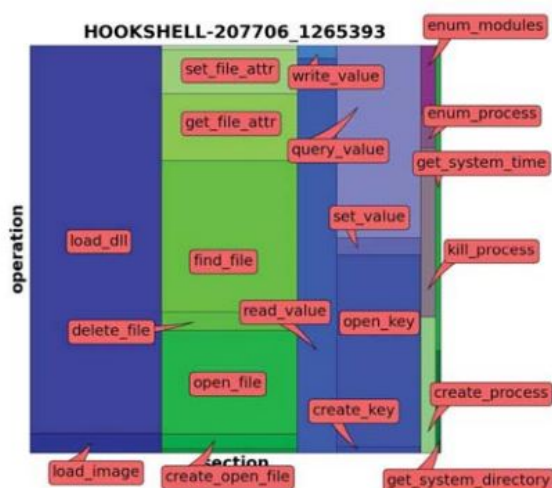


Figure 2.7: Treemap: used system call operations for an individual malware sample [4]

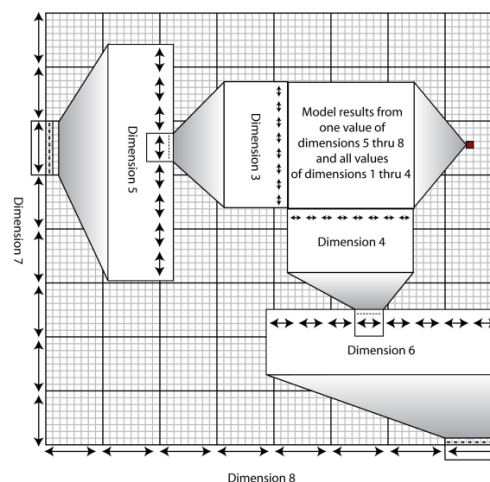


Figure 2.8: Dimensional stacking: projection of high-dimensional data by embedding dimensions within other dimensions [17]

- **Dense pixel displays** map each data point to a colored pixel and these are grouped into adjacent areas that represent individual data dimensions. The classical methods are matrix visualization, circle segment technique, or recursive pattern method (see figures 2.9 and 2.10). This family and more precisely the matrix visualization technique is very appropriate here because each pixel of the visualization is used to store information, the format is convenient for the CNNs and it is very similar to an array of bytes, which is basically an executable. The generation using raw data is very fast and compact. More details on this approach will be given in the chapter 3.

A last question to ask ourselves for the data visualization is whether we need to interact with the visualization. In this work, we choose not to interact with it to speed up the generation time and to simplify the task of the classifier with no loss of precision however. For the same reasons, we keep the visualization in 2D and unique, i.e. only one visualization per malware. So, we will not explore theses features.

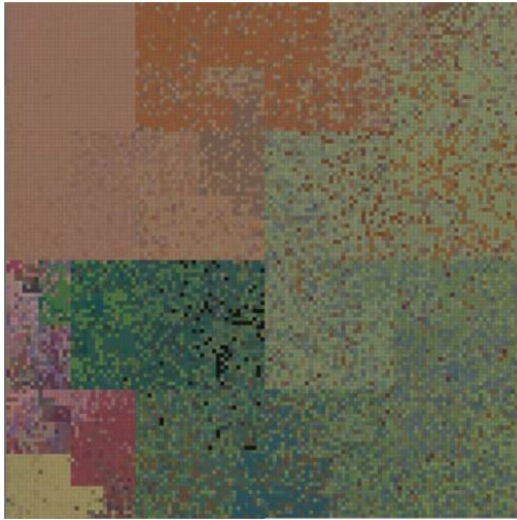


Figure 2.9: Matrix visualization: similar malware samples are arranged next to each other and assigned similar color values to visualize commonalities [18]

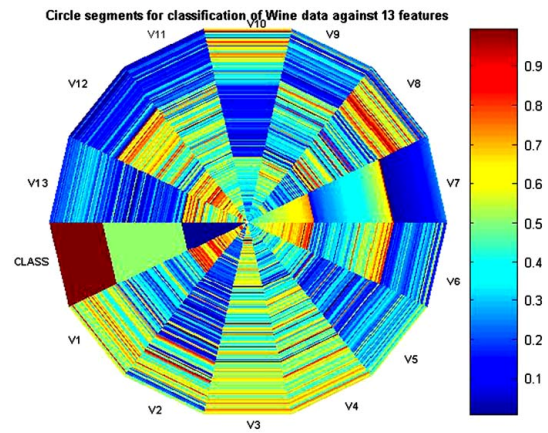


Figure 2.10: Circle segment technique: classification of Wine data against 13 features [19]

2.2 Machine learning and classification

In recent times, the field of machine learning has seen a phenomenal renewed interest with the rise of new methods. Inspired by human biology, these new computational models can have far better performances than the previous algorithms. To better understand these tools, let us go back to the basis of artificial intelligence.

2.2.1 Artificial neural networks

In this section, we will explain what is machine learning and which approaches and models could be interesting to use for the classification in our work.

Artificial intelligence and machine learning

There is no unique definition of what is Artificial Intelligence (AI). For some authors, it is "the automation of activities that we associate with human thinking, activities such as decision-making, problem solving, learning..." (Bellman, 1978). For others, it is "the branch of computer science that is concerned with the automation of intelligent behavior" (Luger and Strubblefield, 1993). For Alan Turing, it is a program able to pass the Turing test (1950): according to Turing, an intelligent behavior is the ability to achieve human-level performance in all cognitive tasks,

sufficient to fool an interrogator [20]. The computer would need to possess the following capabilities: **natural language** processing to communicate with the human interrogator, **knowledge representation** to store information provided before or during the interrogation, **automated reasoning** to use the stored information to answer questions and draw new conclusions and **machine learning** to adapt to new circumstances and to detect and extrapolate patterns [21].

This last capability is of particular interest to us. Machine Learning (ML) is the study of computer algorithms that improve automatically through experience and by the use of data [22]. Machine learning algorithms build a model based on sample data, known as "training data", in order to make predictions or decisions without being explicitly programmed to do so [23].

Approaches and models

There are different types of machine learning algorithms, varying in their approach, the type of data and the type of problem. For the process of classification, the literature widely use supervised learning algorithms. In this approach, we build a mathematical model of a set of data that contains both the inputs (here the visualizations of our malwares) and the outputs (here the true family of these malwares). This set of data is called the training data. Through iterative optimization of an objective function, supervised learning algorithms learn a function that can be used to predict the output associated with new inputs [24]. The goal is to allow the algorithm to correctly determine the output (the family of the malware) for inputs (visualizations of malwares) that were not a part of the training data. To test this way the precision of the algorithm, we use a set of data called the test set. An algorithm that improves the accuracy of its outputs or predictions over time is said to have learned to perform that task [25].

To perform machine learning, we need to create a mathematical model, trained on some training data and then able to process additional data to make predictions. Various types of models have been developed, each with their strong and weak points. The most used and the most efficient model for problems of computer vision with well-supplied datasets is the artificial neural network (ANN).

Artificial neural networks and deep learning

The idea of an ANN is inspired from the biological neural networks that constitute animal brains. They learn to perform tasks by considering examples, generally without being programmed with any task-specific rules. An ANN is a set of nodes called "artificial neurons". Between the nodes, we can have some connections, called "edges", able to transmit information from one node to another. A node can receive

a signal, process it and then send it, modified or not, to other nodes. The signal can be understood as a real number, the process would be a non-linear function of the sum of the input signals, for example, and the output would be the new resulting signal. The nodes and the edges typically have weights that adjust as learning proceeds. The weight controls the strength of the signal and its influence on the next nodes. ANNs sometimes have an activation threshold; the processed signal is transmitted only if the threshold is exceeded.

Typically, the nodes of the ANN are organized into layers. Different layers can have different processing functions and perform different kind of transformations on their inputs. Data runs through the ANN from the first layer (the input layer) to the last layer (the output layer), possibly after traversing the layers multiple times.

ANNs can also use deep learning. Deep learning is a class of machine learning algorithms that uses multiple layers to progressively extract higher-level features from the raw input [26]. The first layers will identify the low-level features and layers after layers, the model will be able to identify higher-level features until being able to identify human concepts such as digits, letters or faces. Figure 2.11 shows the global structure of an ANN with deep learning.

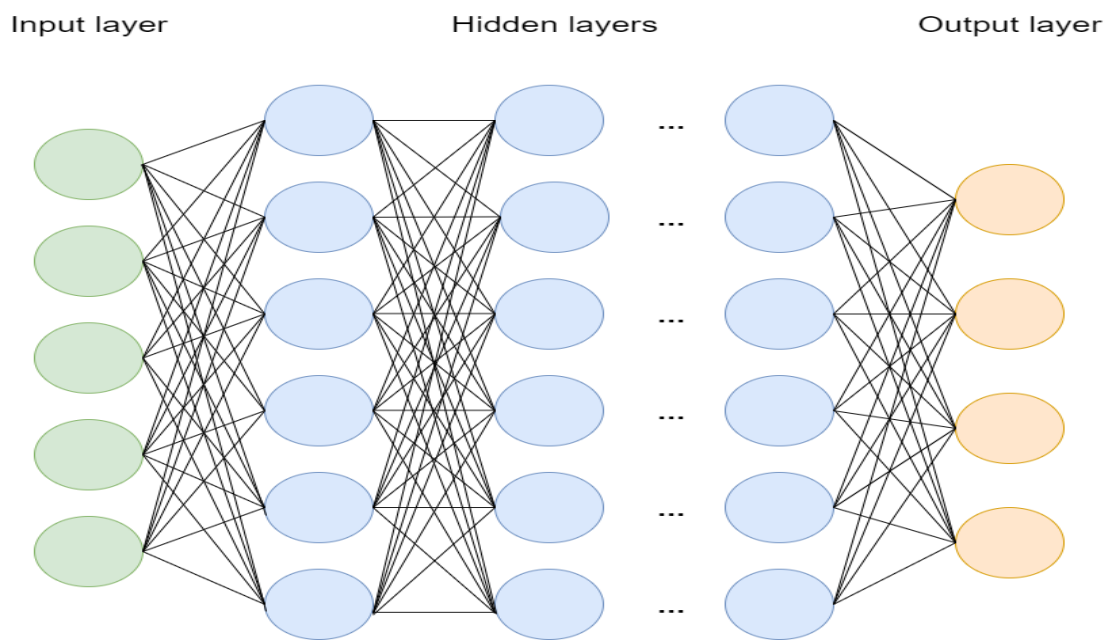


Figure 2.11: Global structure of an ANN with deep learning: an input layer, one or more hidden layers and an output layer; edges connectivity and the number of nodes may vary

2.2.2 Multi layer perceptrons

One type of ANN is the Multi Layer Perceptron (MLP). MLPs can be used to create mathematical models by regression analysis. As classification is a particular case of regression, when the response variable is categorical, MLPs make good classifier algorithms [27]. MLPs contain many perceptrons organized into layers so let us first understand what is a perceptron.

Perceptrons and activation functions

The activation function of an artificial neuron defines the output of that neuron given an input. There exists a lot of possible activation functions, here is a definition of the most used ones. They are represented below on the figure 2.12.

- The **identity** function, also called linear, is the most basic activation function. The artificial neuron always outputs the input x with no modification.

$$f(x) = x \quad (2.1)$$

- The **unit step** function, also called binary step or Heaviside step, consists of a simple threshold: if the input x is positive, it returns 1, otherwise it returns 0.

$$f(x) = \begin{cases} 0, & \text{if } x < 0 \\ 1, & \text{if } x \geq 0 \end{cases} \quad (2.2)$$

The threshold can be adapted to be different from 0.

- Also called soft step or by extension sigmoid, the **logistic** function outputs a softer step, growing slowly from 0 to 1.

$$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}} \quad (2.3)$$

- The **tanh** activation function has a bigger range, growing more quickly but from -1 to 1 .

$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad (2.4)$$

- The **Rectified Linear Unit** (ReLU) function is non-linear and is currently used a lot in deep neural networks because it reduces the likelihood of vanishing gradient, resulting in faster learning, and it uses sparse representations, speeding up the computations and sparing the memory space.

$$f(x) = \max(0, x) = \begin{cases} 0, & \text{if } x \leq 0 \\ x, & \text{if } x > 0 \end{cases} \quad (2.5)$$

All the activation functions presented until here were functions of only one fold from the previous layers, where a fold is a set of, usually consecutive, records of the dataset. Contrary to the next activation function.

- The **softmax** function is a function from multiple folds from the previous layers. Also this function is not represented on the figure 2.12.

$$f(x) = \frac{e^{x_i}}{\sum_{j=1}^J e^{x_j}} \text{ for } i = 1, \dots, J \quad (2.6)$$

where all the x_i are the elements of the input vector and the term on the bottom of the formula is the normalization term which ensures that all the output values of the function will sum to 1.

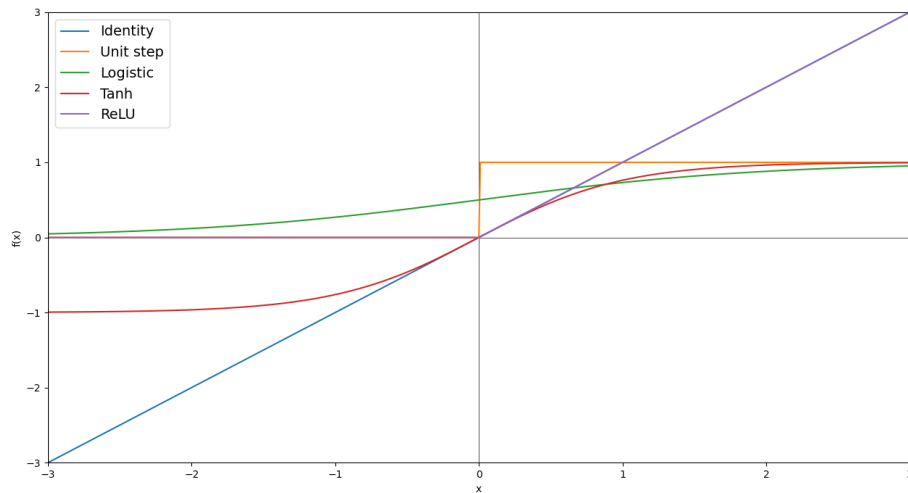


Figure 2.12: Most used activation functions on one fold from the previous layers

In the context of neural networks, a perceptron is an artificial neuron using the Heaviside step function (unit step function) as the activation function, the previously called "process".

Multi layer perceptrons and backpropagation

A MLP is constituted, as it is an ANN, of an input layer, an output layer and some hidden layers. Except for the input nodes, each node can use a nonlinear activation function. So, in the strictest possible sense, MLP "perceptrons" are not perceptrons since they do not use a threshold activation function such as the Heaviside step function. A true perceptron performs binary classification where an MLP neuron is free to either perform classification or regression, depending upon its activation function [28].

All the nodes of an MLP are fully connected between the different layers, each node in one layer connects with a certain weight w_{ij} to every node in the following layer. Learning occurs by changing connection weights after each piece of data is processed, based on the amount of error in the output compared to the expected result. For training, to propagate the error from the last layer, where the prediction is compared to the expected result, MLP uses a supervised learning technique called backpropagation.

The **BackPropagation** (BP) is a generalization of the least mean squares algorithm in the linear perceptron. It uses a loss function, a measure of the difference between the output result and the expected result. The BP algorithm works by computing the gradient of the loss function with respect to each weight by the chain rule, computing the gradient one layer at a time. The algorithm iterates backwards from the last layer to avoid redundant calculations of intermediate terms in the chain rule.

For the n^{th} training example, we can compute the degree of error in the output node j with:

$$e_j(n) = d_j(n) - y_j(n) \quad (2.7)$$

where d is the expected output and y is the real output produced by the perceptron. The node weight can then be adjusted based on corrections that minimize the error in the entire output, given by:

$$\epsilon(n) = \frac{1}{2} \sum_j e_j^2(n) \quad (2.8)$$

Using the gradient descent, a first-order iterative optimization algorithm for finding a local minimum of a differentiable function, we can adjust each weight

layer after layer. The idea is to take repeated steps in the opposite direction of the gradient of the function at the current point. By doing this, we will always move in the direction of the steepest descent. The change in each weight is:

$$\Delta w_{ij}(n) = -\eta \frac{\partial \epsilon(n)}{\partial v_j(n)} y_i(n) \quad (2.9)$$

where Δw_{ij} is the weight variation for edge between node i and node j , y_i is the output of the previous node, v_j is the induced local field, i.e. the output of the summation unit and the input of the next activation function, and η is the learning rate, which is selected to ensure that the weights quickly converge to a response without oscillation.

2.2.3 Convolutional neural networks

Another type of ANN is the Convolutional Neural Network (CNN). The problem of the MLPs is that we only use fully connected nodes, i.e. all nodes of a layer are connected to all nodes of the neighboring layer, making them prone to overfitting. A network overfits when it corresponds too closely or exactly to the training dataset and then can fail to fit additional data or predict future observations reliably. To avoid overfitting, we use regularization techniques such as penalizing parameters during training, skipping connections, dropout, ... CNNs use an additional technique, taking advantage of the hierarchical pattern in data. In a way, CNNs are regularized versions of MLPs [29].

A CNN can have different types of layers between the input and the output. The figure 2.13 shows the typical architecture of a CNN. Each layer has a different goal and a different utility in the training and the prediction of the network. Let us explore them.

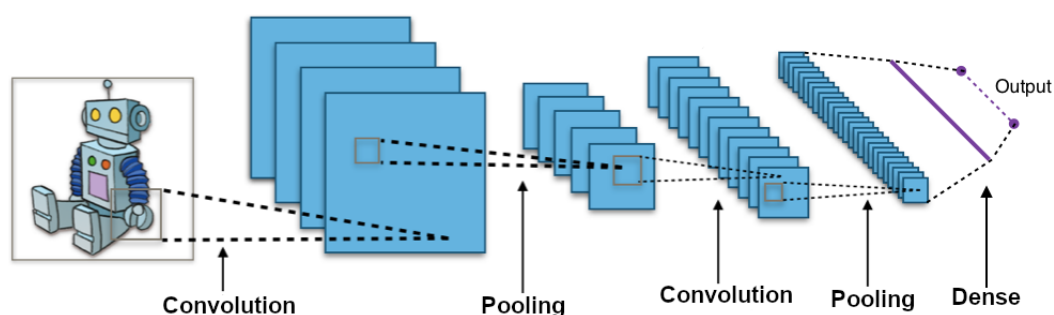


Figure 2.13: Typical CNN architecture [30]

Convolution The core layer of a CNN is the convolutional layer, it is from this layer that the CNN takes its name. First hidden layer of a CNN but also used further, the convolutional layer applies the convolution operation on the input data. After passing through this layer, the image becomes abstracted to a feature map, also called an activation map. This layer consists of a set of learnable filters (or kernels), with small receptive field, but extend through the full depth of the input volume. Each filter computes the dot product between the filter entries and the input, producing a 2-dimensional activation map of that filter. Each neuron receives input from only a restricted area of the previous layer called the neuron's receptive field. Whereas, in a fully connected layer (such as the ones of a MLP), the receptive field is the entire previous layer. Thus, in a succession of convolutional layers, each neuron takes input from a larger area in the input than previous layers. As a result of the convolutional layer, the network learns filters that activate when it detects some specific type of feature at some spatial position in the input [31].

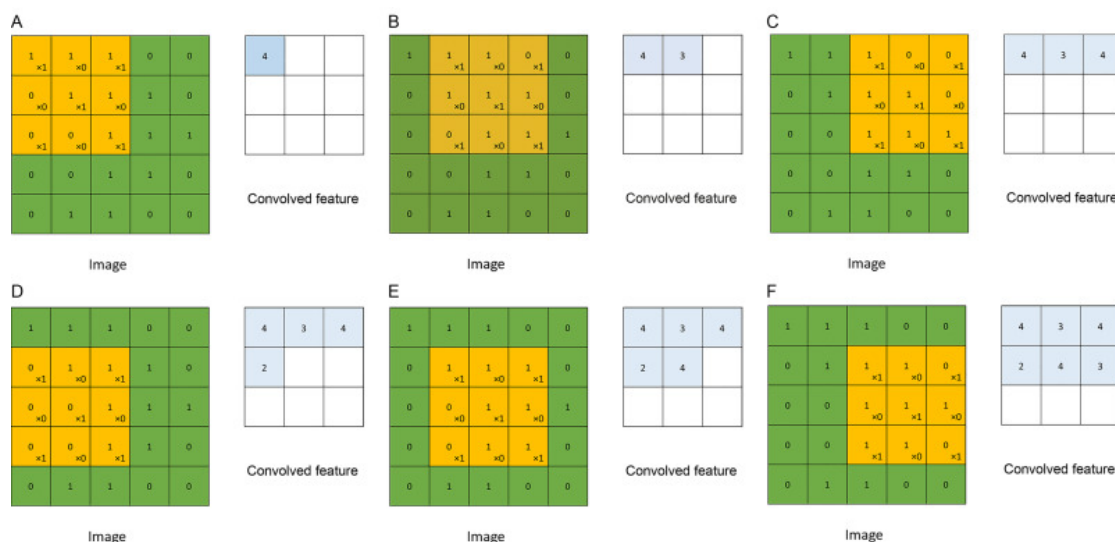


Figure 2.14: Convolution operation [32]

The figure 2.14 shows the application of the convolution operation. As we can observe, the application of the convolution causes a loss of dimension. There is a way to avoid that, called the padding. The idea is simple, we put additional pixels fixed to 0 all around the image and the output of the convolution will have the same size than the input.

The last important parameter of this layer is the stride. It controls how many pixels we move before the next application of the convolution operation. To summarize, the spatial size of the output volume is a function of the input volume size W , the kernel field size K of the convolutional layer neurons, the stride S and

the amount of zero padding P on the border. The number of neurons that fit in a given volume is then:

$$\frac{W - K + 2P}{S} + 1 \quad (2.10)$$

Pooling Another important layer in a CNN is the pooling layer. The pooling layer serves to progressively reduce the spatial size of the representation, to reduce the number of parameters, memory usage and amount of computation in the network, and hence to also control overfitting. It is common to periodically insert a pooling layer between successive convolutional layers, each one typically followed by an activation function, such as those presented in the subsection 2.2.2. The most common form of pooling is the max pooling, we keep only the maximum value of the current filter. Figure 2.15 shows the application of a max pooling layer with a 2×2 filter and a stride of 2. Other pooling functions, such as average pooling, exist but max pooling generally performs better in practice [33].

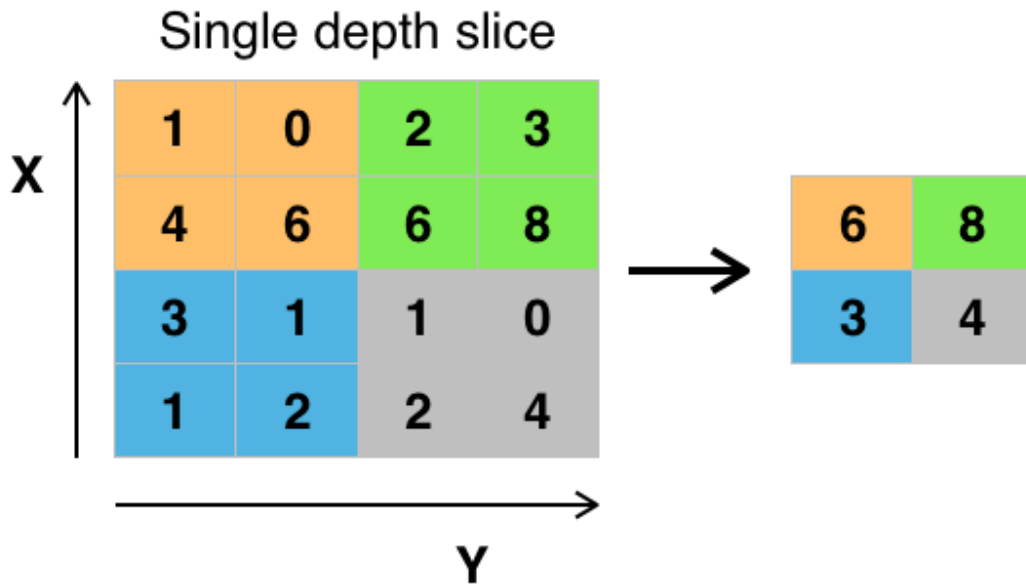


Figure 2.15: Max pooling with filter 2×2 and stride = 2 [34]

After the pooling layer, the commonly used activation function is ReLU. It has the advantage to train faster and without a significant penalty to generalization accuracy.

Dropout A dropout layer randomly sets input units to 0 with a parameter rate at each step during training time, which helps to prevent overfitting. Inputs not set to 0 are scaled up by $1/(1-\text{rate})$ such that the sum over all inputs is unchanged. The dropout layer is not shown on the figure 2.13 since dropout is optional and this layer can be merged at the input of other layers.

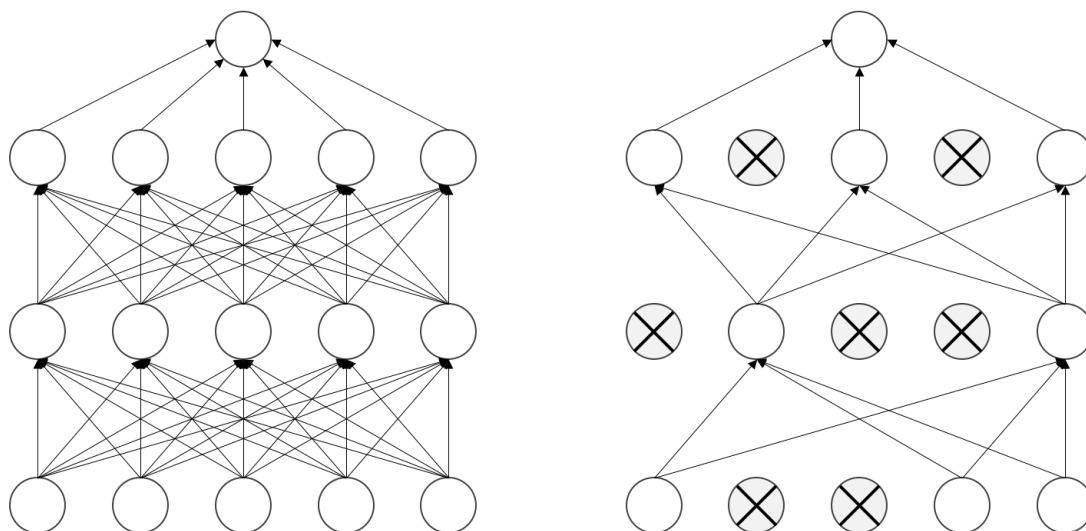


Figure 2.16: Illustration of a neural network before (left) and after (right) the application of a dropout, inspired from [35]

Dense After a succession of convolutional layers and pooling layers, the size of the data is reduced and we can do the final classification with a fully connected layer. So we can see the final part of a CNN as a MLP. As the data is less big and more compacted, this dense layer is less expensive and more precise than a classical MLP. Before the use of the first dense layer, we need to use a flatten operation to put all the data in one dimension of features. A CNN can have several successive dense layers and the last layer often has as many nodes as there are classification families to output a membership probability for each class.

Loss The loss layer or function specifies how training penalizes the deviation between the predicted output and the expected output. It is typically included in the last layer of the CNN. Various loss functions can be used, depending on the specific task. For the prediction of K independent probability values (in our work, K would be the number of malware families), we use the Categorical Cross-Entropy (CCE) function for the loss layer, widely used for multiclass categorization problems. The CCE loss is defined as:

$$CCE = - \sum_i^C t_i \log(s_i) \quad (2.11)$$

where t_i and s_i are the groundtruth and the CNN score for each class i in C . A final activation function, such as softmax, is applied to the scores before the CCE loss computation.

2.3 Summary

We covered the two main data extraction methods, the five big families of visualization techniques and some possible classifiers. Other ANNs exist but MLP and CNN are the most efficient for the image classification. In the next chapters, we will develop the chosen methods, explain their benefits and drawbacks. All the justifications and choices will be illustrated with experiments and results.

Chapter 3

Visualizations

Now that we explored the different data extraction methods and visualization techniques, we are able to choose ours. Let us remember that there is no consensus in the community as to which representation is generally better suited for visualizations [4, 9]. There exists a lot of different techniques, each with their benefits and drawbacks. So our choice will be a compromise between our different importance criteria.

We search for a method able to extract as much information as possible from a sample and transforming it into a visualization as quickly as possible. So in the previous chapter we set our priorities in order of importance as: precision, speed and storage space.

In this current chapter we discuss about how we chose an input format for the CNN and how we built it. A lot of forms are possible and all the choices made are explained here.

3.1 Extraction method

As discussed in the previous chapter, the computer scientists use two main techniques to extract data from unknown samples: static analysis and dynamic analysis. Dynamic analysis is generally slow and complex, providing a plethora of information on the sample. This approach is more appropriate to deepen the analysis of a sample but in our work, we want a fast tool able to analyze the samples as they arrive, for example, at the entry of a network. The dynamic analysis would be really too slow for this purpose. Static analysis therefore seems more appropriate but disassembling the sample code can often slow down the analysis process. So we chose to use a limited static approach which is to generate images directly from the raw sample data.

Introduced by Jianwen Fu et al. [6], this approach is faster since we have no disassembling, execution or preprocessing step. It is multi-extension since all files are made up of bytes regardless of their extension, it avoids the possible detection and evasion techniques and it also processes the encrypted files with no additional complexity.

With this method, we have a risk to lose data information. One goal of this work is precisely to try to obtain a precision similar to a classic static analysis while using a faster generation technique. Such a tool would be ideal for a preliminary scan at the entrance of a high-throughput network or system. Samples classified with less precision could then undergo further analysis with slower but better performing tools.

3.2 Visualization method

Now that we know how to extract the information from our samples, we need to choose how to present the information for the classification. To choose, we will again use our chosen priorities in order of importance: precision, speed and storage space.

The previous chapter introduced the five big families of visualization techniques. We search for a family where the visualization generation is independent of the number of features to preserve a maximum precision for the classification, a family with a simple generation process to ensure the shortest possible generation time and a family with a compact representation to take up less space in memory and favor the use of the CNN convolution operation.

The family that best meets all these criteria is the dense pixel displays family. The idea is to simply map the malware bytes to the pixels in the image. This is an easy operation that uses little computing power while ensuring true precision since the size of the visualization is similar to that of the sample. The storage space is not optimal but acceptable for total preservation of all sample data.

We also chose not to interact with the visualizations, keeping them in 2D, unique and simple to speed up the generation time. The generated images are optimized for machine learning classification and are therefore not intended to be easy for humans to interpret. Other tools exist for this purpose such as *PE Studio*¹ or *Wireshark*².

¹<https://www.winitor.com/>

²<https://www.wireshark.org/>

3.3 Generation of the visualizations

In malware comparison and classification, there are two possible approaches: the feature-based approach [7] which requires more computing time to extract the features and the image-based approach [8] that we chose here for its simplicity and its speed. A big weakness of this approach is that our visualization can be dominated by some eventual embedded images, as will be shown later in this chapter. In this case, we would need to use the feature-based approach, disassemble the sample and lose a lot of speed. However, the image-based approach stays a good first step in a classification pipeline and if not decisive, usage of more expensive tools is still possible.

3.3.1 Mapping

In this work, we directly use the raw file to generate a matrix visualization of the dense pixel displays family. For that, we map each byte of the file to a pixel of the visualization since a byte has a value between 0 and 255 just like a pixel on one color channel. We can see each file as an array of bytes and each image as an array of pixels.

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00005E10 C3 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC AAAAAAAAAAAAAA
00005E20 8B 54 24 0C 8B 4C 24 04 85 D2 74 47 33 C0 8A 44 <TS...ÖtG3A5D
00005E30 24 08 57 8B F9 83 FA 04 72 2D F7 D9 83 E1 03 74 S.WüfÜ.r--Üyá.t
00005E40 08 2B D1 88 07 47 49 75 FA 8B C9 C1 E0 08 03 C1 .H".GiuöcEÄä..A
00005E50 8B C9 C1 E0 10 03 C1 8B CA 83 E2 03 C1 E9 02 74 <EÄÄ..ÄkEjÄ..Äe.t
00005E60 06 F3 AB 85 D2 74 06 88 07 47 4A 75 FA 8B 44 24 .öeÖt..".GjuöcD$
00005E70 08 5F C3 8B 44 24 04 C3 55 8B EC 51 8B 45 08 8D .ÄkD$.ÄÜkLQkE..
00005E80 48 01 81 F9 00 01 00 00 77 0C 8B 0D 80 F8 40 00 H..Ü...v..e.eö8.
00005E90 08 B7 04 41 B8 52 8B C9 56 8B 35 80 F8 40 00 C1 .".ÄskEöcSeö8.A
00005EA0 F9 08 0F B6 D1 F6 44 56 01 80 5E 74 0E 80 65 FE Ö..BÖÜV.e't.eö8
00005EB0 00 88 4D FC 88 45 FD 6A 02 EB 09 80 65 FD 00 88 .Mu"Öyj..e.eöy."
00005EC0 45 FC 6A 01 58 8D 4D 0A 6A 01 6A 00 6A 00 51 50 Eöj.X.M.j..j..QP
00005ED0 8D 45 FC 50 6A 01 B8 F8 31 00 00 83 C4 1C 85 C0 .EÜFj.eö1..fA..A
00005EE0 75 02 C9 C3 0F B7 45 0A 23 45 0C C9 C3 68 00 00 u.EÄ..E..E..EÄh..
```

Figure 3.1: The hexadecimal representation of a sample (in the red rectangle) [36]

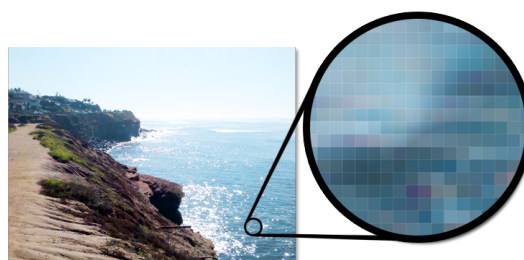


Figure 3.2: A zoom to the pixels of a picture [37]

Since the behavior of this type of malware visualization has never been analyzed under the classification of a CNN in the literature, we used three different visualization techniques to compare the results.

3.3.2 Size of the visualizations

Since each malware has a different size and that we map the bytes of each malware to the pixels of an image, the size of these output images will be different. For more coherence, we arbitrarily fixed some widths for the images according to the size of the malware file. As for the height, no size is fixed to avoid losing information by

restricting yourself to a defined format. This choice is often made in the literature [8, 38]. Here are our widths restricted depending on the sample size.

Size of the sample [bytes]	width [pixels]	height [pixels]
0 - 10.239	32	unlimited
10.240 - 30.720	64	unlimited
30.721 - 61.440	128	unlimited
61.441 - 102.400	256	unlimited
102.401 - 204.800	384	unlimited
204.801 - 512.000	512	unlimited
512.001 - 1.024.000	768	unlimited
1.024.001 - ...	1.024	unlimited

Table 3.1: Fixed sizes of the visualizations according to the size of the samples

These values have no influence on the precision of the CNN since it propagates the information to the neighborhood, progressively reducing the resolution of the visualization. So, if the image is bigger than the input value of the CNN, we will propagate the pixels overage to the neighbors to resize the visualization while keeping information about the whole image.

3.3.3 Used visualizations

To compare different behaviors with a CNN, we chose to use three different visualizations. We want to compare the speed of generation and the precision in the classification. Here are the three different visualizations used and analyzed in this work.

Grayscale visualization

The first method simply maps the byte values to a one channel array of pixels (0 is black and 255 is white), inspired from Nataraj et al. [11]. The resulting visualizations are gray images. We can already see, on the figure 3.3, some differences between the visualizations from multiple malware families. More visualization results are available in the Appendix A.

RGB visualization

The second method is original to this work. It uses the same principle as the first method but the images use a three channels array of pixels. The algorithm takes the bytes of the sample, three by three, and put them in the channels of the image:

Red, Green and Blue (RGB) (see figure 3.4). This visualization is more compact than the previous one and globally takes the same memory space.

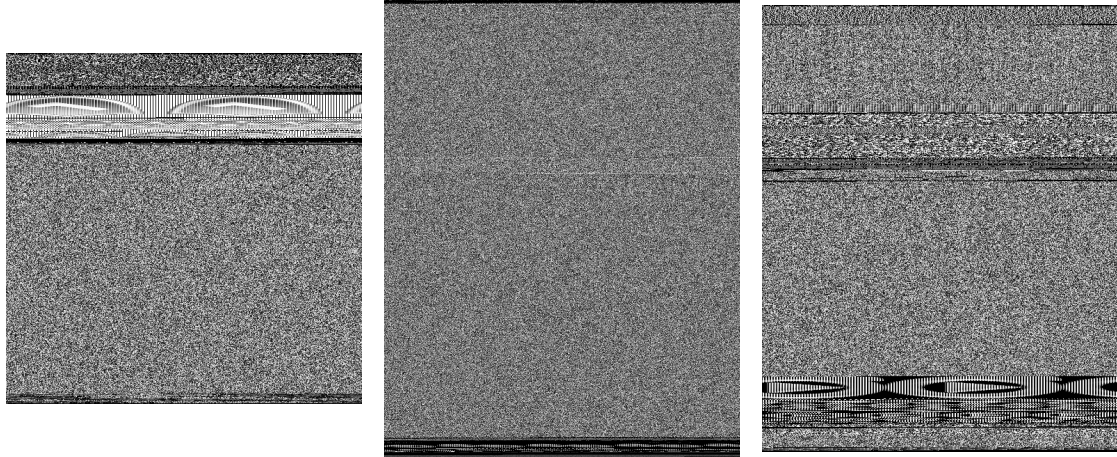


Figure 3.3: Grayscale visualizations from 3 different malware families: 1clickdownload (SHA256: 00a23ccd 9a348812 00c269bf dbf9b293), ainslot (SHA256: 00e947cc 0d0cc9c0 f46dbf9d d5cd33b9) and allapple (SHA256: 00023154 6b0e5eb4 306b31c6 81b4d366) (from left to right)

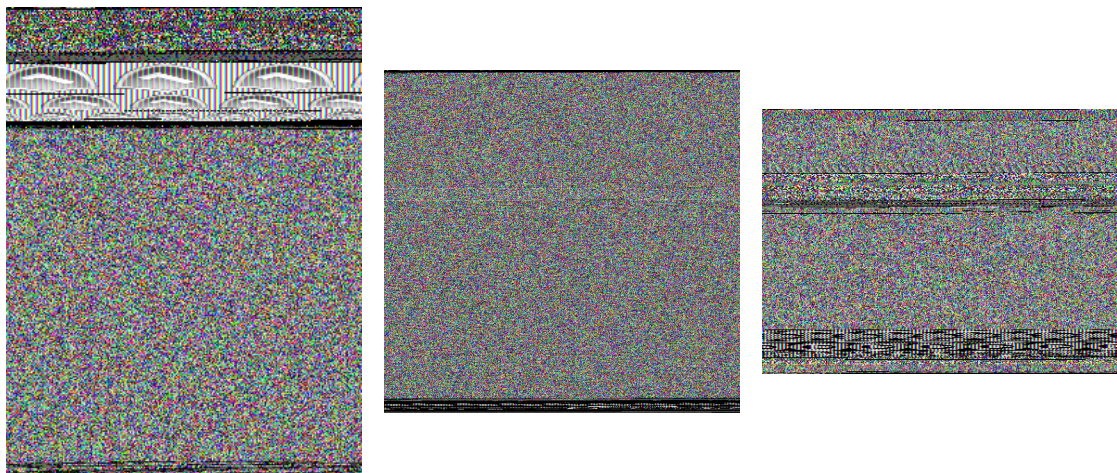


Figure 3.4: RGB visualizations from three different malware families: 1clickdownload (SHA256: 00a23ccd 9a348812 00c269bf dbf9b293), ainslot (SHA256: 00e947cc 0d0cc9c0 f46dbf9d d5cd33b9) and allapple (SHA256: 00023154 6b0e5eb4 306b31c6 81b4d366) (from left to right); some visualizations may appear distorted due to the layout

RGB tuned visualization

The third method, inspired by Fu et al. [6], also uses a three channel array of pixels but their content is a little different. Here are the formulas for each channel.

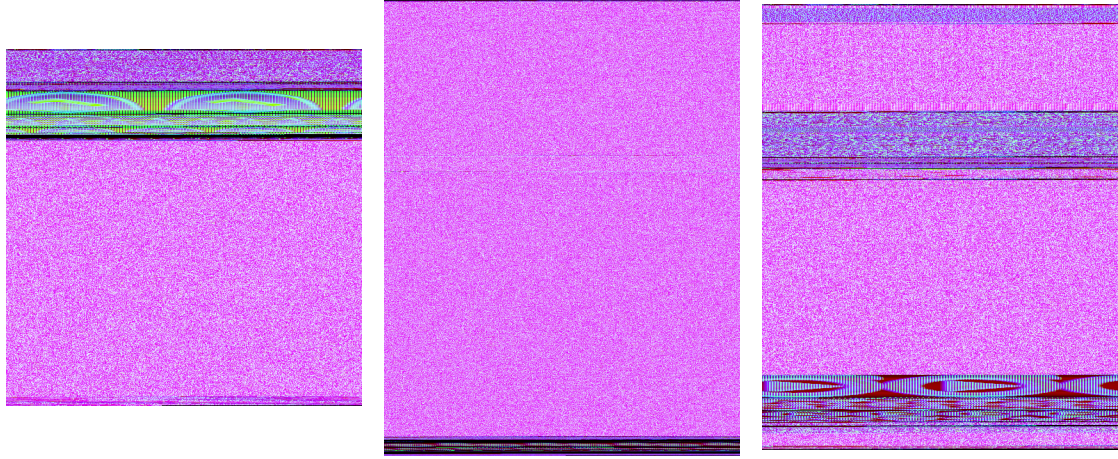


Figure 3.5: RGB tuned visualizations from three different malware families: 1click-download (SHA256: 00a23ccd 9a348812 00c269bf dbf9b293), ainslot (SHA256: 00e947cc 0d0cc9c0 f46dbf9d d5cd33b9) and allapple (SHA256: 00023154 6b0e5eb4 306b31c6 81b4d366) (from left to right) (from left to right)

For the **Green** channel, the algorithm simply maps the value of the byte to a value in the channel, as the grayscale visualization does.

For the **Red** channel, we compute the global entropy of the next section of 256 bytes. Entropy can represent a measure of disorder of byte values in the section, it can demonstrate high-level characteristics and reflect the similarities of the malware in general. Entropy can be calculated as:

$$RedComponent = \left(- \sum_{i=0}^{255} p(c_i) \log_2 p(c_i) \right) \times \frac{255}{8} \quad (3.1)$$

where $p(c_i)$ is the probability of occurrence of byte value i . When all of byte values in a section are the same, the entropy obtains the minimum value of 0; in contrast, the entropy obtains the maximum value of 8 when all the byte values are different. Then, the entropy is scaled to the range $[0;256]$ of the red color channel.

For the **Blue** channel, we replace the relative size of the current section used in Fu et al. [6], unavailable in our representation with no disassembling, by an entropy on a smaller section of 8 bytes. The idea is to reflect a more local behavior in the bytes of the sample. This local entropy can be computed as:

$$BlueComponent = \left(- \sum_{i=0}^8 p(c_i) \log_2 p(c_i) \right) \times \frac{255}{8} \times 85 \quad (3.2)$$

We obtain visualizations, on figure 3.5, with a shape similar to the first method but we gain a lot of information with the color of the pixels. Let's decompose an image to understand the effect and the meaning of each channel on the final image.

3.3.4 RGB tuned: channels decomposition

To better understand which information is contained in each channel for the RGB tuned visualization, we will decompose by color a malware from the amonetize family. We can see on the figure 3.6 the original visualization generated from the malware and the figure 3.7 shows the three different channels separately.



Figure 3.6: RGB tuned visualization from the amonetize (SHA256: 00faa8a87f86db20 ab8c9011 3889118e) malware family

We can see that the green channel just maps the bytes to the pixels and we have the same result as if we had used the grayscale visualization but in green. The red channel with the global entropy is more uniform since it depends on the next 256 bytes. The sudden changes in byte value in a fairly uniform region in terms of byte value are smoothed out in the red channel giving a fairly constant red color. On the other hand, the long zones with a change of byte value are highlighted by a

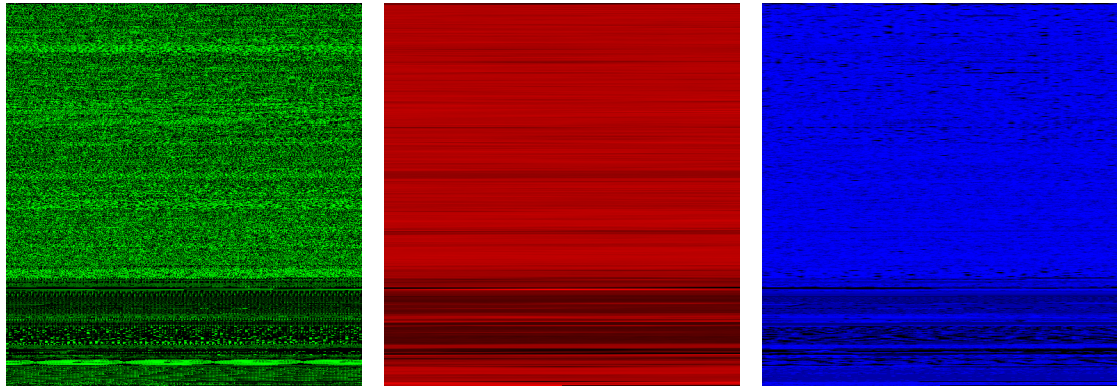


Figure 3.7: RGB tuned visualization from the amonetize (SHA256: 00faa8a87f86db20 ab8c9011 3889118e) malware family decomposed by color channel

red of a different color. The red channel therefore highlights areas with close byte values from neighbor to neighbor over a long data period. Finally, the blue channel adds more precision with the local entropy and a more local variation of the bytes value. It highlights areas with close byte values from neighbor to neighbor over a shorter data period, reflecting the proximity variations.

So, the total image simultaneously reflects the whole static code of the sample, the global variation of behavior along the file and the local variation of behavior along the sections.

3.3.5 Embedded images

As cited previously, a big weakness of our extraction approach is that our visualization can be dominated by some eventual embedded images, i.e an image used in the code sample. Indeed, we can see on the figure 3.8 that about a third of the sample visualization is occupied by embedded images which will not be especially useful for sample classification.

We could remove the embedded images from the samples with an automated tool, it would not specially be easy to do that and the problem would not be solved. In fact, the problem is twofold. If we remove the embedded images and we use these samples to train our CNN, it will not recognize for sure the sample with the embedded images since a significant part will be different. Indeed, images often take up more storage space than malware code so an attacker could hide his small malware in a lot of big images. Conversely, training CNN with the embedded images will not teach the CNN to recognize the sample alone. So, we need to stay aware that drowning malwares in embedded images can be an attack technique to deceive a CNN with our visualization technique.

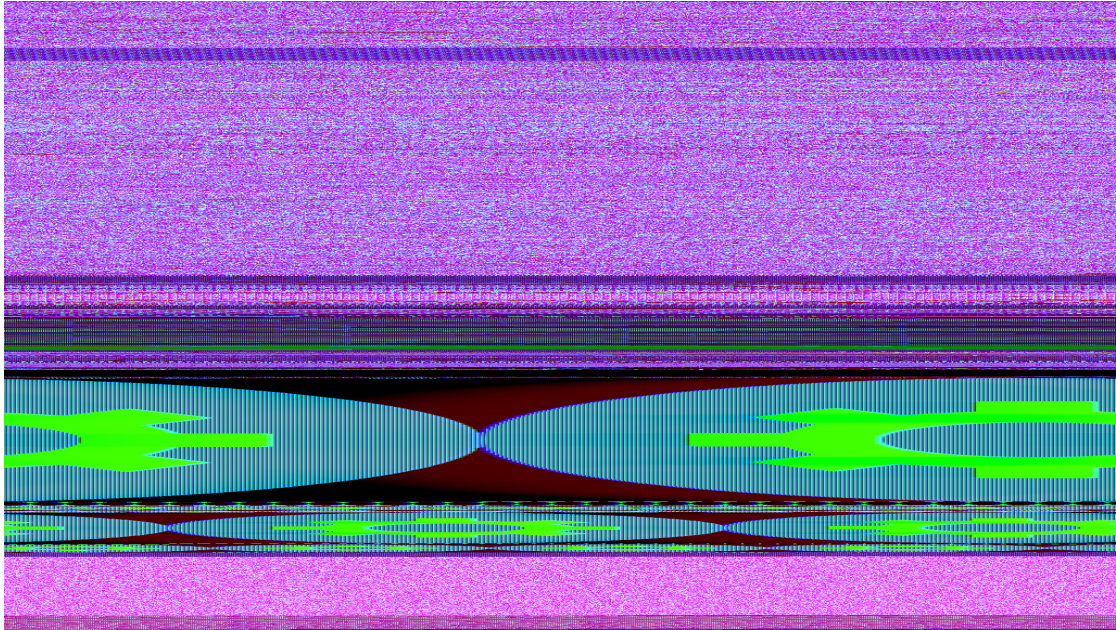


Figure 3.8: RGB tuned visualization from the autoit (SHA256: 011c7cb1 7d6d0af5 604a8fc2 c2bb34f5) malware family with an embedded image

Now that we chose our extraction technique and that we built different visualization techniques, making us aware of the functioning and the risks of these methods, we can be interested in the classification and in the architecture of our MLP and CNN.

3.3.6 Study of family diversity

We presented the images of three malwares from different families with our three visualization techniques (see figures 3.3, 3.4 and 3.5). Now let us take a look at the images of different malwares from the same family. The goal is to have a more precise idea of the visual diversity possible within a class. For this purpose, we chose to use, as an indicator, the malware family "blackmoon" with the RGB tuned visualization method.

The figure 3.9 shows three different "blackmoon" malwares with very similar visualizations. On the contrary, the figure 3.10 shows three other "blackmoon" malwares with very different visualizations. For these, we can even see that the size of the files is not at all the same.

We can say that, at least for the human eye, these visualizations are very different and we probably would not group them together in the same family if we were asked to classify them. But the most important thing is that the computer is

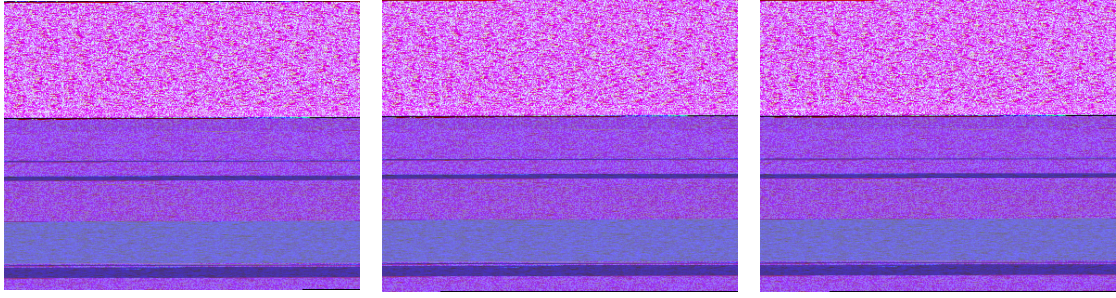


Figure 3.9: Similar visualizations of three malwares from the blackmoon family (from left to right, SHA256: 002f34b3 60261bda 7f691a3a 96aa83e0, SHA256: 004b731e bfce328e 10c0bfe8 7e9fa872 and SHA256: 004c0f4f 6d6cbd75 31fba8ae9834fe2)

able to interpret these images and derive features to learn in order to recognize the family later. We just need to have enough similar images for each diversity in each family to correctly train the classifier.

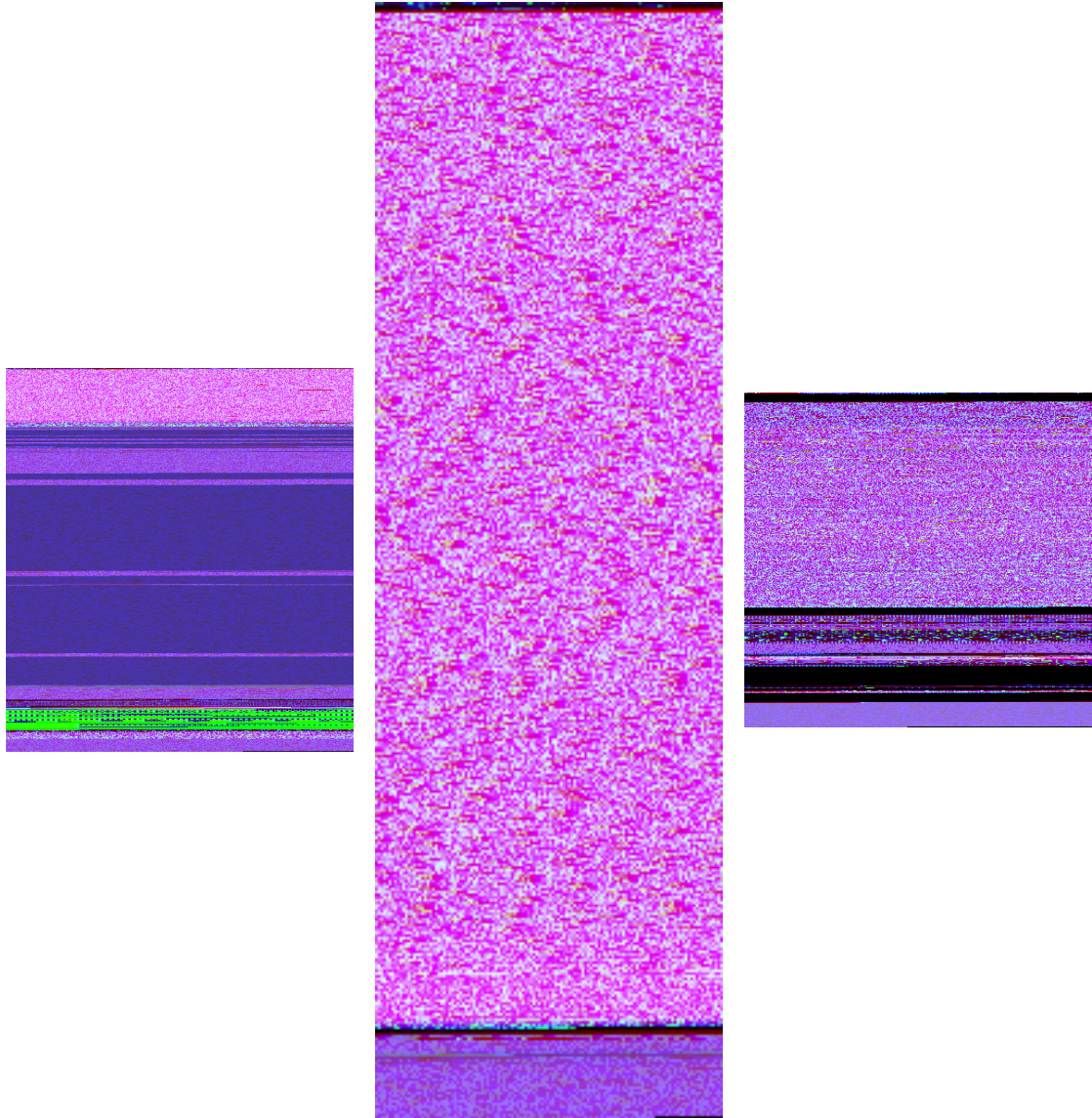


Figure 3.10: Different visualizations of three malwares from the blackmoon family (from left to right, SHA256: 005c24d4 fb89d9c1 3677f203 55f3fe4a, SHA256: 00a9263f 2a056e62 631e0437 b80dd482 and SHA256: 0075c094 f5cbe6f1 f4949b33 09508558)

Chapter 4

Classification

We covered the background needed to understand the MLP and the CNN and we chose our visualization methods, setting all the parameters for the input format of our classification process. Now let us choose the architecture of our classifier and the global structure of its layers.

Let us remember that this work does not follow the classic way of malware analysis, decomposing the sample to recognize it or learn more for future analyses. Here, we take the raw data of the sample and we transform it into an image. So we need a classifier especially efficient for dealing with images. This classifier must be as precise as possible while being fast enough to be able to process the entry of a high-throughput network or system.

We chose to use a CNN as classifier in this work for its processing quality with the images and its speed. To have a point of comparison, we also implemented a MLP to show the interest of the CNN and the possible gains of this newer classifier.

4.1 Our architectures

Let us first explore the global architecture of our classifiers. The sequence of layers will be explained here and the experiments for the parameters fitting will be presented in the next chapter.

4.1.1 MLP

Our MLP has been implemented to have a reference to which we can compare the results of our CNN, but we still optimized its parameters. It is constituted of 4 successive dense layers with a dropout layer inserted and followed by a compilation with a CCE loss function.

The order of layers is:

- dense layer with variable number of nodes
- dense layer with variable number of nodes
- dropout layer with variable parameter rate
- dense layer with variable number of nodes
- dense layer with as many nodes as there are families in the dataset

Let us remember that the last layer has as many nodes as there are families in the dataset to output a membership probability for each family. This order of layers has been chosen to have a sufficiently developed MLP to allow efficient operation but not too many layers to avoid making the model too complex which would risk losing precision. Indeed, too many layers means a lot of parameters to train and therefore a complex and slow model, difficult to train. However, this organization is not fixed: in the next chapter we will fit the parameters using experiments and from an experience to another we will withdraw one layer or another.

4.1.2 CNN

Our CNN is the main tool of this work, we expect good results from it. Its architecture is inspired from numerous architectures in the literature [39, 40]. It consists of an alternation of convolutional and pooling layers, followed by successive dense layers after a flatten layer and with dropout layers inserted. The model is finished by a compilation with a CCE loss function.

The exact succession of layers is:

- 2D convolutional layer
- max pooling layer
- 2D convolutional layer
- max pooling layer
- dropout layer with variable parameter rate
- flatten layer to put the data in one dimension
- dense layer with variable number of nodes
- dropout layer with variable parameter rate

- dense layer with variable number of nodes
- dense layer with as many nodes as there are families in the dataset

Again, this organization is not fixed: in the next chapter we will fit the parameters using experiments and from an experience to another we will withdraw one layer or another.

4.2 Dataset apportionment

As discussed in the chapter 2, our datasets need to be divided into a training set and a testing set. The training data will be used to teach the model to recognize the correct family for the samples using backpropagation to adjust the weights and the testing data will be used to evaluate the precision of the model. In reality, the training set is itself divided into training data used to train and validation data used to evaluate the data to see if we still need to train it or to avoid overfitting.

To split our data into training and testing sets, we followed the idea of Guyon et al. [41]. If we call N the number of families of recognizers, h_{max} the largest complexity of those families, f the testing set size and g the training set size, the ratio f/g scales like:

$$\sqrt{\frac{\ln N}{h_{max}}} \tag{4.1}$$

Using this formula on our main dataset of 102 families which will be presented in the next chapter, we obtain an apportionment of 70% for the training set and 30% for the testing set. For the other datasets, we are not so far up to 5% different. For simplicity, we chose to keep this ratio of 0.7/0.3 not so far from the Pareto principle [42], saying that 80% of consequences come from 20% of the causes.

For the cross-validation, we chose to use the same ratio, so the training set is again divided in a ratio of 0.7/0.3 for the training data and the validation data. This choice is sufficient to ensure a good training for our datasets of around 10,000 samples and not too much to avoid overfitting and to allow normalization.

4.3 Choice of fixed parameters

Before we run the experiments to set the parameters, let us start by exploring the fixed parameters and why we decided to set them this way.

4.3.1 Epochs

In machine learning, an epoch indicates the number of passes of the entire training dataset the model has completed. Datasets are sometimes grouped into batches, some small subsets of the dataset, to run with less data at the same time but we decided to keep all the dataset in one batch so the number of epochs is the number of iterations.

We chose arbitrarily to set the number of epochs to 20 but we configured an early stopping with a patience of 2. It means that if after 2 successive epochs the loss value is not decreasing, the training process stops. This configuration avoids overfitting, preventing the model to increase its loss function that we want as small as possible.

4.3.2 Input shape

The input shape of the MLP and the CNN have been set to (224, 224, 3) to take into account the biggest possible windows of data, using the 3 color channels. Experiments of the figure 4.1 show that to decrease the size of this window decreases the precision of the model and figure 4.2 shows that to increase the size makes the model slower. So this choice is a compromise between speed and precision.

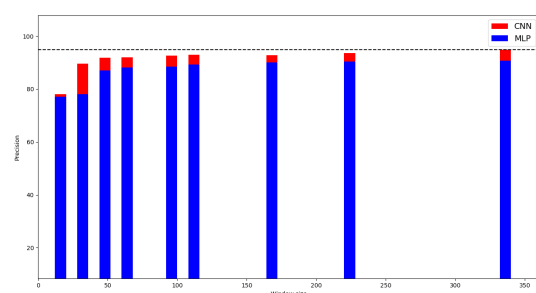


Figure 4.1: Variation of precision in percentage at the entry of MLP and CNN according to input shape

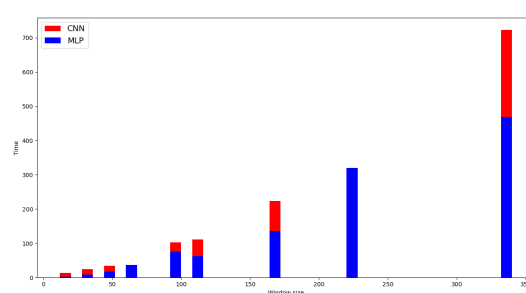


Figure 4.2: Variation of time in seconds at the entry of MLP and CNN according to input shape

These experiments were carried out with a dataset of 1996 images following the chosen dataset apportionment and the 20 epochs with early stopping. All other parameters are fixed but not yet optimal, so it is possible to obtain better results later in our work.

We could take a bigger windows size but it makes the model more complex with more parameters and it requires a powerful equipment to train it. The best precision executable on our computer is (224,224,3).

4.3.3 Activation function

On each convolution and dense layers, we use an activation function. For these layers we chose to use the ReLU function. This non-linear function is currently the most used function with the CNN. It reduces the probability of having a noisy gradient, resulting in faster learning and better precision. It also speeds up the computations and reduces the memory space needed. The figures 4.3 and 4.4 clearly show the advantages of ReLU compared to the other activation functions.

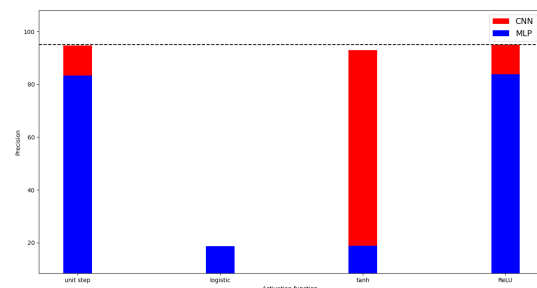


Figure 4.3: Variation of precision in percentage of MLP and CNN according to activation function

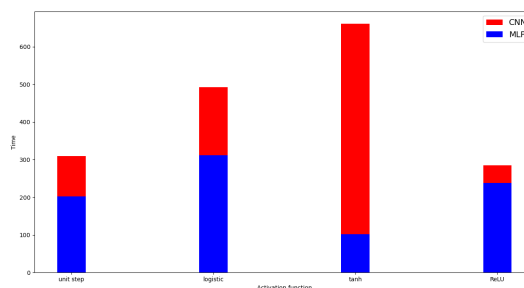


Figure 4.4: Variation of time in seconds of MLP and CNN according to activation function

We also use a different output activation function on the last dense layer of the model. For this layer, we chose to use the softmax function because it is a function from multiple folds from the previous layers so it takes into account all the layers which is good for a last layer. The figures 4.5 and 4.6 show the advantage of softmax on the other final activation functions.

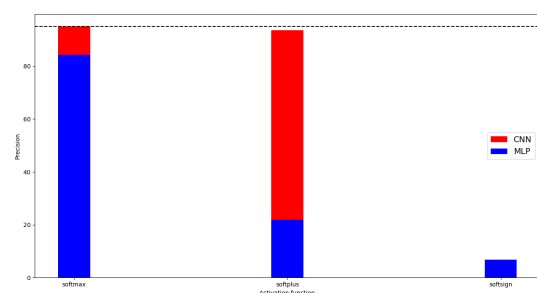


Figure 4.5: Variation of precision in percentage of MLP and CNN according to final activation function

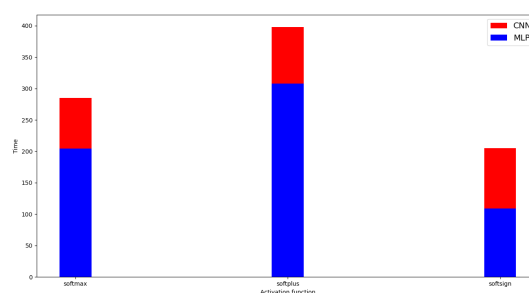


Figure 4.6: Variation of time in seconds of MLP and CNN according to final activation function

4.3.4 Padding

As discussed in the chapter 2, the application of the convolution operation causes a loss of dimension and padding allows to avoid that. By adding additional pixels set to 0 all around the image, the output of the convolution will have the same size than the input, which is preferable to keep consistency in the model and it allows to better take into account the value of the side pixels of the image. We tested different parameters on the CNN, with and without padding, and the accuracy increases up to 3% with padding and the accuracy never decreases with padding.

4.3.5 Metrics

A metric is a function that is used to judge the performance of the model. Metric functions are similar to loss functions, except that the results from evaluating a metric are not used when training the model. We chose to use different metrics in this work to have different approaches on the performances of the model.

We use classification *accuracy*, computed as the total number of correct predictions divided by the total number of predictions, to have a first good idea of the success or not of the model. Mathematically, we write:

$$Accuracy = \frac{True\ Positive}{True\ Positive + False\ Positive} \quad (4.2)$$

where a true positive is a malware of the class, correctly classified as such and a false positive is not a malware of the class but it is misclassified as such.

We also use *recall* which is slightly different from accuracy and computed as the total number of correct predictions divided by the total number of true malwares. Recall is a more significant measure if we want to secure a network flow for example since it is less fault tolerant of accepting malware as a friendly sample. Indeed, rejecting a friendly sample is less problematic than accepting a malicious sample. Mathematically, we write:

$$Recall = \frac{True\ Positive}{True\ Positive + False\ Negative} \quad (4.3)$$

where a false negative is classified as not being a malware of the class when it is.

Finally we use confusion matrix which shows for each family in the classification process how the samples have been predicted. Each row of the matrix represents the samples in an actual family while each column represents the samples in a predicted class. As we will see in the subsection 5.4.3, it is easy to see whether the

system is confusing two classes, commonly mislabeling one as another. This tool will be very useful later in this work.

4.3.6 Loss function

The loss function quantifies the cost of the model and lets the model know if it is going in the right direction by minimizing the loss function at each step. There exists a lot of different loss functions. Figures 4.7 and 4.8 compare the accuracy and the time of the main ones on our MLP and CNN. We chose to use the Categorical Cross-Entropy (CCE) which has the best score of accuracy and a correct speed.

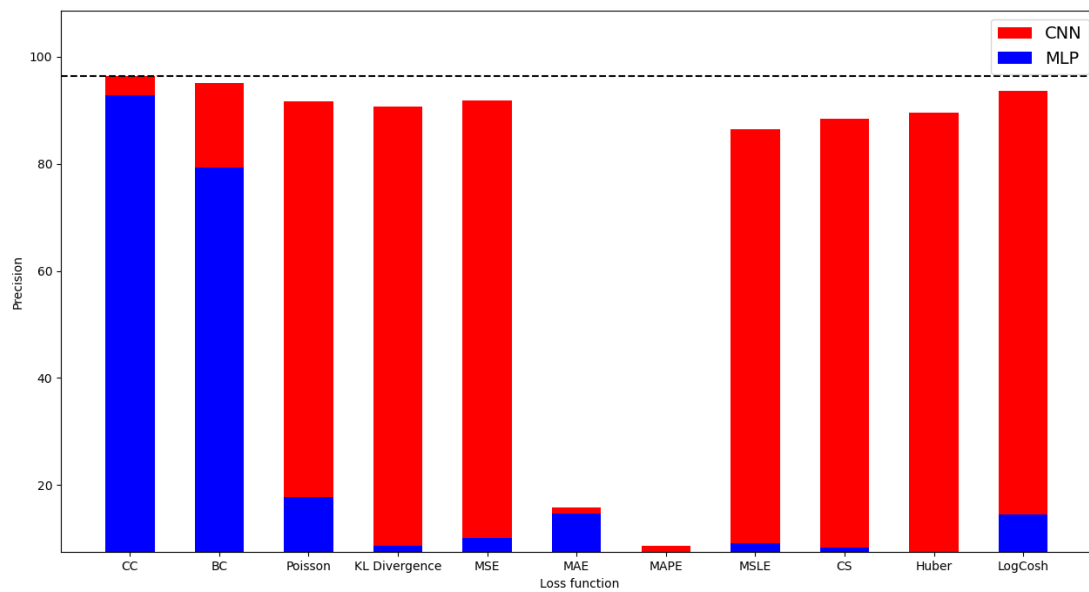


Figure 4.7: Variation of precision in percentage of MLP and CNN according to loss function

4.3.7 Optimizer and learning rate

Where the loss functions are the mathematical way of measuring how wrong the predictions are, the optimizers are there to tie together the loss function and model by updating the model in response to the output of the loss function. The loss function is the guide to the terrain, telling the optimizer when it is moving in the right or wrong direction.

With the optimizers, a very important parameter is the learning rate. Changing our weights too fast by adding or subtracting too much can hinder our ability to

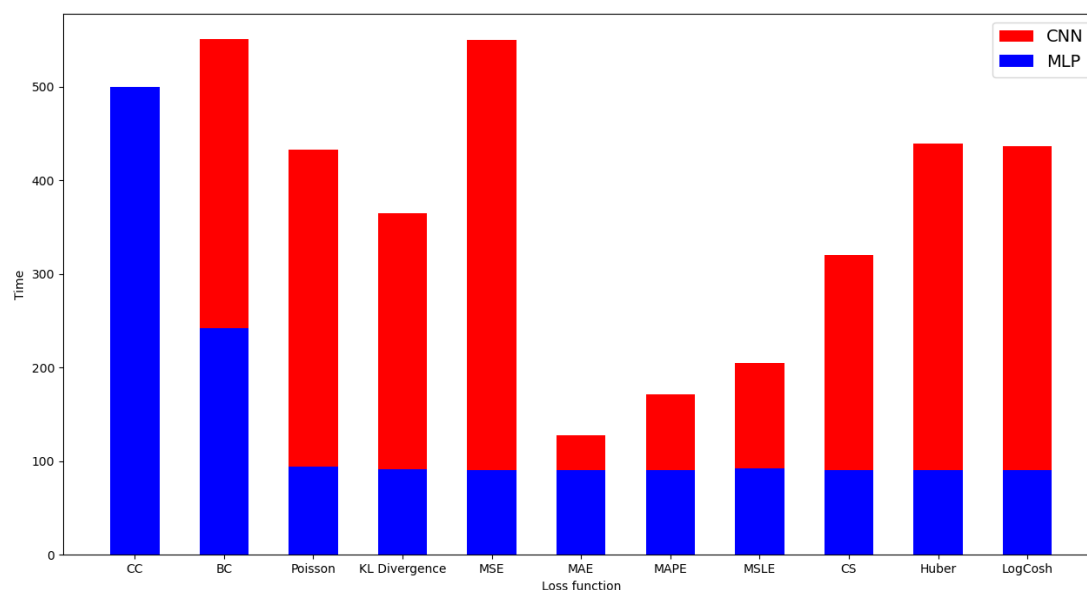


Figure 4.8: Variation of time in seconds of MLP and CNN according to loss function

minimize the loss function. We do not want to make a jump so large that we skip over the optimal value for a given weight and we do not want to make a jump too small and get stuck in a local minimum. To make sure that this does not happen, we use a variable called the learning rate. It is just a number that we multiply the gradients by to scale them in the gradient descent algorithm. This number and the way we make the gradient descent depend on the optimizer chosen [43].

The figures 4.9 and 4.10 compare different optimizers and their accuracy and speed. We chose the Adam optimizer in this work for its accuracy and its speed. Adamax could have been another good choice with a better speed and a slightly lower precision but our first criteria of choice is the accuracy and Adam is not so slow.

Adam optimization is a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments. It has recently seen broader adoption for deep learning applications in computer vision and natural language processing. With Adam, the gradient descent maintains a single learning rate for all weight updates and the learning rate does not change during training, which makes it straightforward to implement and computationally efficient [44].

Now that we set up the fixed parameters of our classifiers, it is time to conduct experiments to reach the best performances and find the most interesting visualization method. The next chapter will present the different datasets used, compare the different visualization methods, optimize the parameters for the best results

and discuss the obtained results.

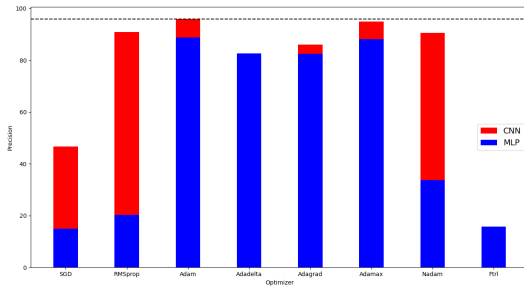


Figure 4.9: Variation of precision in percentage for MLP and CNN according to optimizer

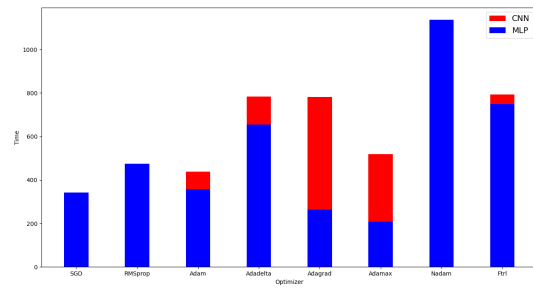


Figure 4.10: Variation of time in seconds for MLP and CNN according to optimizer

Chapter 5

Experiments and Results

In this chapter, we will use the three different visualization methods introduced in the chapter 3 and the MLP and CNN introduced in the chapter 4 to tune their parameters and try to have the better accuracy possible. We will compare the results with the literature and offer the best model of this work.

5.1 Presentation of the datasets

Let us begin by introducing the different datasets we used and by analyzing their constitution to better understand the results of the experiments on them.

5.1.1 Cisco dataset

This dataset has been constituted by EPL researchers, using samples from Cisco collected between April and June 2020. It contains 13 059 malwares from 102 different families. The malwares have been labeled thanks to the VirusTotal API and the use of AVClass [45]. The different families of all datasets are detailed in the appendix B.1.

The figure 5.1 shows the malware distribution among families. We can see disparities among families, some barely have a few viruses while others have several thousand. With some malware background we can also see that some families are very similar to others, containing the same types of viruses with very similar behavior. For example, for downloadassistant and downloadguide or for installcore and installbrain, we can besides notice that their visualizations have points in common. These information will be useful to analyze the classification later in this chapter.

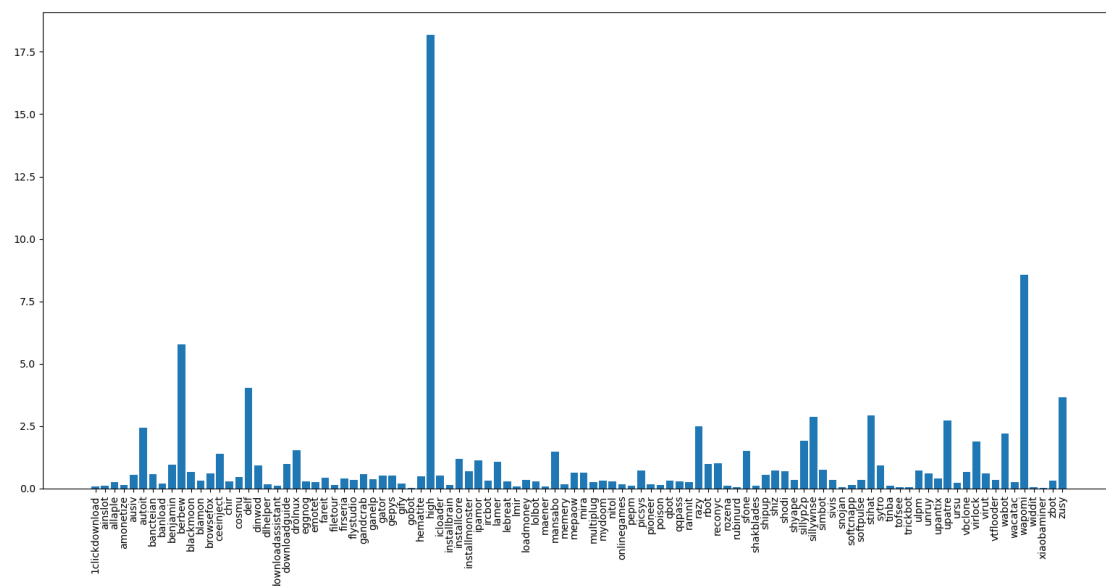


Figure 5.1: Percentage of malwares by family in the Cisco dataset

5.1.2 Microsoft malware classification challenge on Kaggle

This dataset is on free access on Kaggle¹. Microsoft organized a challenge of malware classification into families based on file content and characteristics [46]. This dataset is no more really "fashionable" because it dates from 2015 but many research papers use it to evaluate their performances so it is a good point of comparison for us [8, 7]. It contains 10,868 malwares from 9 different families.

The figure 5.2 shows the malware distribution among families. We can see that this dataset is also not very well distributed but we have a smaller number of families for a similar number of malwares.

5.1.3 Cleanwares

Until here, we only have malwares but it would be beneficial if our classifier could also distinguish the malwares from the cleanwares as an anti-virus software do. We found no paper in the literature using classification with visualization on a dataset mixing malwares and cleanwares, so the results will be very exciting. This dataset is constituted from 668 clean softwares with no families, in one class. This dataset comes from the free website Zenodo².

¹<https://www.kaggle.com/c/malware-classification/overview>

²<https://zenodo.org/record/2528209#.YJQf0x9xdEZ>

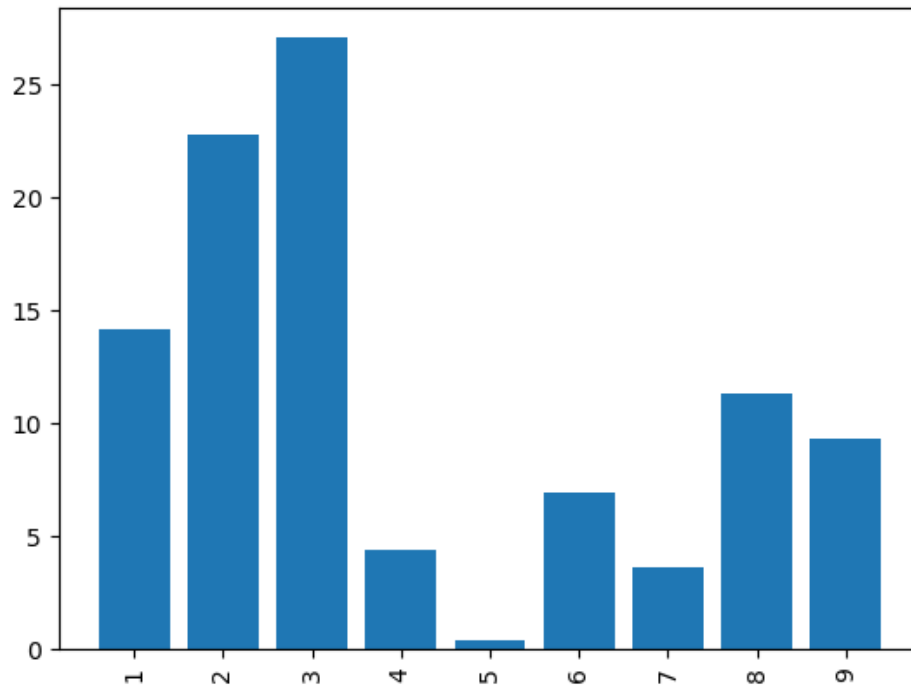


Figure 5.2: Percentage of malwares by family in the Kaggle dataset

5.1.4 Custom dataset

In the idea to compare our results with the paper of Fu et al. [6], from which our third visualization method is inspired, we created a dataset with 15 families and well distributed malwares, as the one used in the paper. We used a subset of the families and malwares from the first introduced dataset, the Cisco dataset. We were unable to obtain the original dataset used in the paper so we used this solution to have a point of comparison. This paper is important to know if our third visualization technique, RGB tuned, is interesting or not.

The figure 5.3 shows the malware distribution among families. We can see that this dataset is already more balanced than the previous ones.

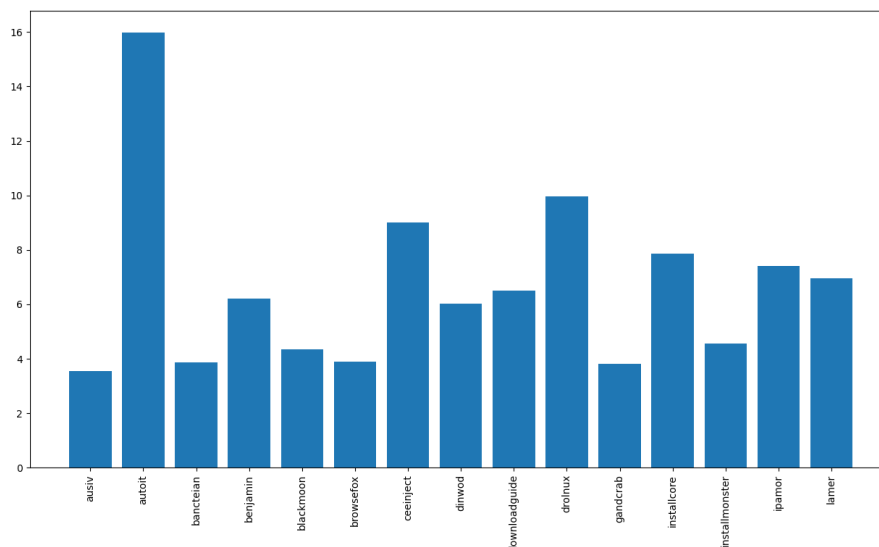


Figure 5.3: Percentage of malwares by family in the custom dataset

5.2 Implementation

To better understand the experiments and to put the results into perspective, it is interesting to analyze the hardware configuration and the Python implementation of the model.

5.2.1 Hardware configuration

All the experimental results obtained in this work were obtained with a laptop Dell G5 with Intel Core i7 8th generation processor (6 x 2.20 GHz), 16 GB DDR4 RAM and Nvidia GeForce GTX 1050 Ti graphic card.

5.2.2 Python implementation

All the Python implementation is available on GitHub³. All the visualizations have been generated using the Python library *PIL*. We used multi-threading to speed up the execution time. For the classification process, we used the Python libraries *keras*, *tensorflow* and *sklearn*. We also used more scientific libraries as *numpy*, *pandas* or *seaborn* and *matplotlib* for the plots.

³https://github.com/Tioneb88/TFE_Malware_Visualization_and_Classification

5.3 Generation time of visualizations

Before discussing about the best visualization method or the best classifier in terms of precision, let us look at the time we need to generate each of our visualization methods, on each of our datasets. To measure the time, we defined three experiments: one with one sample of the dataset of about 580 Ko to compare the global speed, a bigger one with 100 samples to see if it is purely the creation of a new file which takes time and the biggest one with 100 Mo of samples to see if it is the size of the files which takes time to generate.

The visualization methods compared are the three presented in the chapter 3, grayscale, RGB and RGB tuned, and we added a fourth visualization method. This method is used as a point of comparison, it is the method detailed in Fu et al. [6] of which we rewrote the code since it was not available in open access.

The table 5.1 shows the results of these experiments. We can see that as expected the most complex methods take more time so the fastest is the grayscale visualization closely followed by the RGB visualization. Then comes the RGB tuned visualization and finally the slowest is the disassembled visualization. We can conclude that our visualizations are faster, which was to be expected since they do not require either to disassemble the sample or to run it, we will see if in terms of precision it is also a success.

	Grayscale			RGB		
	1 sample	100 samples	100 Mo samples	1 sample	100 samples	100 Mo samples
Cisco dataset	0.233	18.544	22.676	0.259	29.640	37.653
Kaggle dataset	0.306	33.835	24.199	0.269	75.472	37.063
Cleanwares	0.174	10.156	21.732	0.330	16.205	37.061

	RGB tuned			Disassembled		
	1 sample	100 samples	100 Mo samples	1 sample	100 samples	100 Mo samples
Cisco dataset	41.182	7468.964	9127.451	43.906	7692.094	9337.009
Kaggle dataset	30.105	8723.722	5643.717	31.697	8894.341	6112.697
Cleanwares	42.986	4565.152	24658.779	43.504	4667.637	24697.196

Table 5.1: Time in seconds for the generation of visualizations on several datasets: above grayscale and RGB, and below RGB tuned and disassembled

5.4 Classification experiments

Using the presented datasets, our three visualization methods and our classifiers, we carried out multiple experiments leading to several interesting results. Each

of the following subsections corresponds to an experience and a desired goal. The results are then analyzed before moving on to the next experiment.

5.4.1 Comparison of classifiers

Let us remember that we built two different classifiers: a MLP and a CNN. The reason that drove us to develop a MLP is so that it can serve as a benchmark for our CNN. Indeed, our work is a classification with a CNN on visualizations. So it would be a shame to find out that our MLP is outperforming our CNN. Let us find out now.

The table 5.2 compares our MLP and our CNN for different sets of parameters on our first dataset, the Cisco one with 102 families, with the RGB tuned visualization method. We obtain the same global trend with the other visualization techniques but RGB tuned presents the best results for this experiment. As presented in chapter 4, the last dense layer is always with as many nodes as there are families in the dataset, so 102 in this case. We vary the number of nodes for three level of dense layers and the parameter rate of the dropout layers, 2 dropout layers for the CNN and only one, the first, for the MLP. A "/" in the column means that this layer was inactive for the related experiment.

Dense 1	Dense 2	Dense 3	Dropout 1	Dropout 2	MLP accuracy	CNN accuracy
/	256	128	0.25	0.5	37.77	86.22
/	256	128	0.5	0.5	42.69	87.65
/	512	256	0.25	0.5	72.59	88.23
/	512	256	0.5	0.5	74.79	86.32
/	1024	512	0.25	0.5	78.76	86.80
/	1024	512	0.5	0.5	68.61	86.09
1024	512	256	0.25	0.5	71.82	85.63
1024	512	256	0.5	0.5	34.55	77.05
2048	1024	512	0.25	0.5	77.64	85.83
2048	1024	512	0.5	0.5	19.75	86.37
4096	2048	1024	0.25	0.5	73.71	84.92
4096	2048	1024	0.5	0.5	27.84	86.68

Table 5.2: Comparison of accuracy in percentage between our MLP and our CNN on the Cisco dataset, for variable parameters of the dense and dropout layers, using RGB tuned visualization method

We have carried out other experiments but without better results than those presented here. We see that in all cases the CNN obtains a better precision than the MLP and that the best precision, in bold in the table 5.2, is higher with the CNN than with the MLP. So we have proven the added value of using a CNN in this work and all the following experiments will use our CNN.

5.4.2 Comparison of visualizations accuracy

We found our best classifier in terms of accuracy and we found our best visualization method in terms of generation time. We are now interested to see which visualization technique achieves the best classification accuracy with our CNN.

The table 5.3 shows the results of the experiments made with our CNN on the custom dataset for the three visualization methods. We vary the dropout rates with the values 0.25, 0.5 and 0.75 and for each of the possible combinations, we vary the dense layer parameters with five combinations for two layers: 256 – 128, 512 – 256, 1024 – 512, 2048 – 1024 and 4096 – 2048. For ease of reading, we have summarized the variation of the parameters of the dense layer by an interval of precision results, delimited by the lowest and the highest result obtained with these parameters.

Dropout 1	Dropout 2	Grayscale accuracy	RGB accuracy	RGB tuned accuracy
0.25	0.25	90.65 - 92.82	92.15 - 94.99	88.48 - 94.16
0.25	0.5	89.82 - 93.16	91.49 - 94.99	91.15 - 96.82
0.25	0.75	90.15 - 94.16	90.32 - 94.33	90.65 - 94.49
0.5	0.25	90.48 - 94.66	90.48 - 94.16	86.48 - 94.16
0.5	0.5	88.31 - 93.16	91.32 - 95.16	90.82 - 93.32
0.5	0.75	89.15 - 92.15	92.15 - 95.33	89.82 - 93.82
0.75	0.25	91.49 - 93.49	92.15 - 94.99	89.48 - 95.66
0.75	0.5	89.15 - 95.66	91.15 - 94.66	90.48 - 94.99
0.75	0.75	89.82 - 93.16	91.49 - 94.66	90.48 - 93.66

Table 5.3: Comparison of accuracy in percentage for our three visualization methods on the custom dataset, for variable parameters of the dense and dropout layers, using our CNN

From the table 5.3 we learn different things. First, by calculating the average accuracy of the 45 experiments carried out, we obtain an average of 91.96% for grayscale, 93.01% for RGB and 92.45% for RGB tuned. So, on average, RGB is the most promising method. Secondly, it is contrariwise RGB tuned which obtains the best maximum precision, in bold in the table, all experiences combined. Third, if we look for each experiment, which method got the highest score and do the counts, grayscale gets 10 wins, RGB gets 20 wins and RGB tuned gets 15 wins.

It can therefore be concluded that RGB tuned seems to be the best in terms of accuracy to be tuned for stable use. But on average, RGB is the most reliable in accuracy. By using what we know about the generation times of the visualizations, we can therefore say that RGB is the best compromise for a system with an arrival of new samples at high speed. It is quick to generate and efficient to classify. In

any case, grayscale is outdated, lagging behind for precision and not compensating with its generation speed.

Since we are trying to achieve the best possible accuracy, next experiments will mainly use RGB tuned but we know that RGB is the best compromise between speed and accuracy for high-throughput systems.

5.4.3 Comparison with Kaggle

The advantage of working with a public dataset like Kaggle’s is that we can compare our results with other people and that our experiences are fully reproducible with our code, which is also open access. The figure 5.4 shows the leaderboard of the competitors of the Microsoft challenge by order of best loss minimization.

1	▲ 5	say NOOOOO to overfittttting	  	0.00283
2	▲ 7	Marios & Gert	 	0.00324
3	▲ 11	Mikhail & Dmitry & Stanislav	  	0.00396
4	▲ 13	Ivica Jovic		0.00447
5	▲ 8	Octo Guys	 	0.00519
6	▲ 12	Oleksandr Lysenko		0.00533
7	▲ 1	Silogram		0.00632
8	▼ 7	SSIR	   	0.00670
9	▼ 4	gphilippis		0.00681
10	▲ 10	Gilberto Titericz Junior		0.00684

Figure 5.4: Leaderboard of the Microsoft malware classification challenge on Kaggle [46]

We conducted experiments with our three visualization methods on our CNN with different parameters. Our best results are 90.65% for grayscale, 92.39% for RGB and 93.82% for RGB tuned. This best result gives a loss result of 0.013232, ranking us in the top 30.

We might think that this position is not great but let us not forget that we are only using the raw sample data to generate our images. The best in the ranking will surely have used more in-depth but also more time-consuming methods. For our approach it is already a good result.

If we take a look at the literature, Copycat [8] uses this dataset and reaches a maximum 93.34% accuracy while using a sort of grayscale visualization with disassembling and Ni et al. [7] obtains a maximum 98.862% accuracy while extracting the opcodes with disassembled code to generate grayscale images.

On the figure 5.5 we can see the confusion matrix showing the number of well classified and misclassified malwares by family.

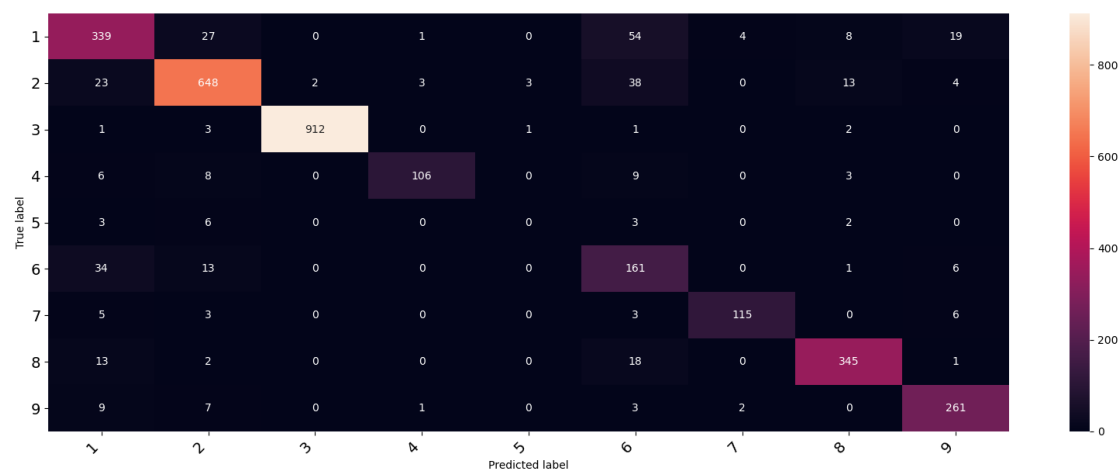


Figure 5.5: Confusion matrix of the classification with CNN on Kaggle dataset

We can notice that malwares of family 1 and 2 are often confused with family 6 and vice versa. By tuning our model more, we could take this misclassification into account more and improve our score. We could adjust the training weights to increase learning for these families or use data enrichment methods to get a larger training set for these families. However, this would limit our model to this dataset, preventing it from being effective on another distribution of families. The goal of our work being to remain polyvalent, we will not try to obtain better results on this dataset but this comparison remains encouraging.

5.4.4 Comparison with literature

Remember that our third visualization method is inspired by Fu et al. [6]. We have shown that our visualization techniques are faster in section 5.3 but now it is time to see if our visualizations provide a better classification accuracy score or not.

Fu et al. prides itself on obtaining the highest 97.47% accuracy on a dataset of 15 families. By reconstructing a dataset of 15 families with our custom dataset, we put our CNN to the test with RGB tuned and we obtained a maximal 97.82% accuracy, beating their classifier and speed at the same time.

5.4.5 Analysis and dissection of Cisco dataset

The Cisco dataset is our biggest dataset but also the most diversified and unbalanced with its 102 families. The highest accuracy obtained on this dataset with our CNN is 88.23% with RGB tuned, which is quite acceptable. At random, we would have a one in 102 chance of correctly classifying a sample. So, we might think that the more families we have, the harder the classification will be. We can also think that the balance of the number of malwares per family will also play a role.

Based on this result, we wonder if it is indeed due to the large number of classes and if gradually reducing the number of classes would increase the accuracy of CNN at the same rate. The figure 5.6 shows our results. We begin on the right with 102 families and we finish on the left with 2 families, reducing by step of 5 families at a time. We decided to remove the families in the order of the most distant number of malwares from the mean number among the families. So, the families with the most unbalanced number of malwares in the dataset were removed first.

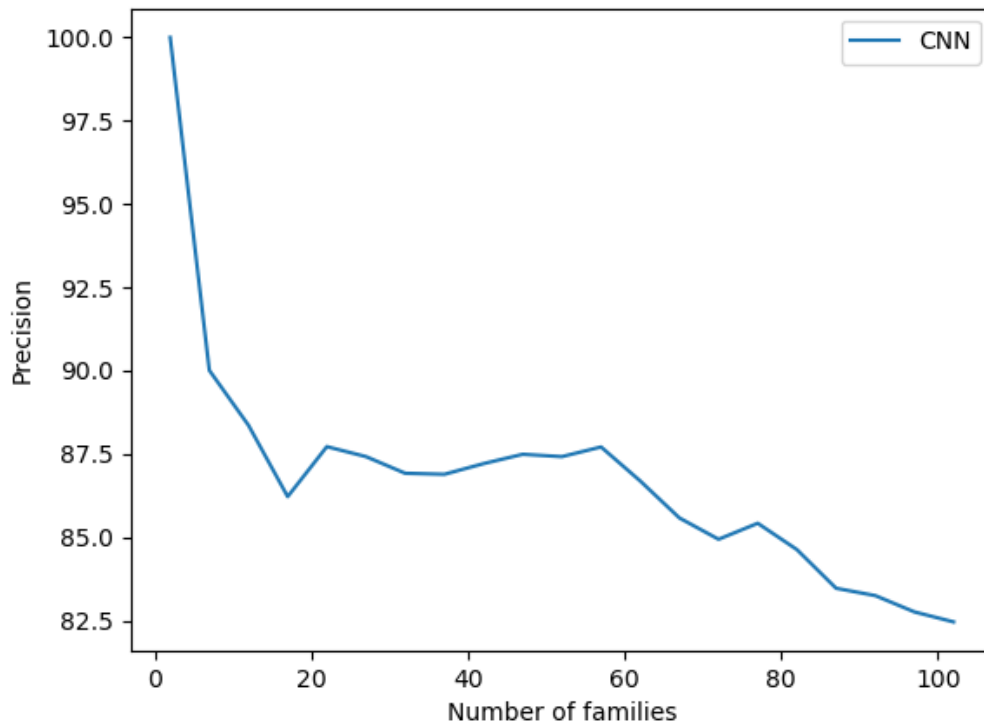


Figure 5.6: Percentage of accuracy of the CNN on the Cisco dataset with variable number of families

The figure 5.7 shows the evolution of the total number of malwares in the dataset as families are withdrawn and the figure 5.8 shows the evolution of the average number of malwares per family as families are withdraw.

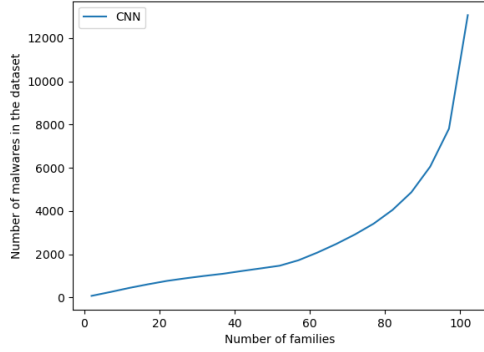


Figure 5.7: Number of malwares in the Cisco dataset with variable number of families

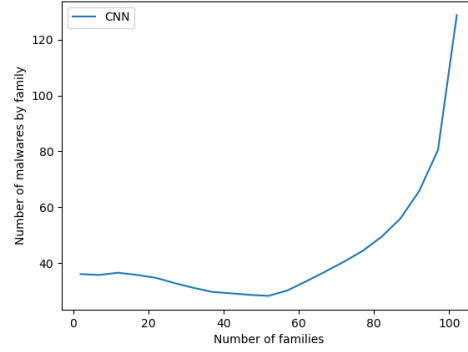


Figure 5.8: Mean number of malwares by family in the Cisco dataset with variable number of families

We observe that the precision increases with the reduction in the number of families but not with a constant rate as we thought. We have a step between 60 and 20 families approximately with no gain in accuracy. This can be explained by the low average number of malware per family at this step. Indeed, we have on average about 30 malwares per family, which does not guarantee an effective training of our CNN and which could explain our poor results.

If we had a more balanced dataset in terms of the number of malwares per family, like the custom dataset, we could hope for better results. Or if we had an imbalanced dataset but with more populated families, our experience of reducing the number of families gradually might be more meaningful.

5.4.6 Classification with cleanwares

We got 88.23% accuracy with our CNN with RGB tuned images from the Cisco dataset and its 102 malwares families. We could ask ourselves if our classifier would still be as efficient by integrating an additional family of cleanwares. We have therefore added a 103rd family to our dataset, made up of our 668 cleanwares presented earlier.

The highest accuracy obtained on this dataset with our CNN is 86.32% with RGB tuned, which means that 61% of our cleanwares are well classified. Recall is a little less with 85.99. It is complicated to quantify this result, especially since the literature does not investigate the subject much and no paper is reproducible.

This result can be considered disappointing since 88% of the malwares is well classified and only 61% of the cleanwares are. But let us remember that here we only have one class of cleanwares where all types of softwares are mixed up, which is not ideal for learning CNN. By using a cleanware dataset as well structured in families as that of our malwares, we could improve the learning of our classifier and expect to obtain better results. Unfortunately, we did not find any dataset of this type.

Chapter 6

Conclusion and further work

In this work we presented two new methods of malware visualization, RGB and RGB tuned. We used the raw sample data to generate dense pixel displays and we used them as input for a MLP and a CNN. These new visualization techniques turned out to be faster to generate than those of the literature disassembling the code. Additionally we presented a CNN as classifier able to achieve higher accuracy on our visualizations than literature. Finally we analyzed the behavior of the classifier with different datasets, exploring their structure to explain the results and we obtained encouraging results by adding cleanwares to the classification.

To conclude, we developed a tool able to quickly analyze and classify a high-speed flow of new samples, for example to secure the input of a network. This tool could be used to do preliminary analyzes and the most doubtful samples could then be analyzed by more complex tools. This tool could therefore save the time of cybersecurity specialists to focus on new threats.

Our method can achieve quick and accurate classification but there are a few limitations to our solution. Our method takes the raw sample data and it can therefore be generalized to any type of file but we need a training dataset similar to the targeted samples to be efficient. So, our approach can be deceived by malwares integrated in big embedded images or by encrypted or packed samples not included in the training dataset. Donahue et al. [47] seems like a good lead to investigate this problem, using a Markov Byte Plot (MBP) based on a Markov random field to visualize packed malware. This approach could be generalized for embedded images.

Our visualization methods are already fast but they could surely be accelerated using more time-saving computing methods. Entropy takes a very long time to compute, we could maybe explore other features to fill the color channels while preserving a good classification accuracy.

Finally future work could focus on the optimization of the classifier parameters. Lately, a lot of new approaches for CNN [48, 49, 50] have been proposed and it would be interesting to try them to further improve the accuracy of our solution. It would allow, for example, to explore new optimizers for stochastic gradient descent, a weighted adaptive learning network to better account for disparities in families or new training strategies for visual classification problems.

Bibliography

- [1] Michael Sikorski and Andrew Honig. *Practical malware analysis: the hands-on guide to dissecting malicious software*. no starch press, 2012.
- [2] J Hernandez-Castro, A Cartwright, and E Cartwright. “An economic analysis of ransomware and its welfare consequences”. In: *Royal Society open science* 7.3 (2020), p. 190023.
- [3] Sam Cook. *Malware statistics and facts from 2021*. comparitech, 2021. URL: <https://www.comparitech.com/antivirus/malware-statistics-facts/>.
- [4] Markus Wagner et al. “A Survey of Visualization Systems for Malware Analysis”. In: *Eurographics Conference on Visualization (EuroVis) ; STARs - State of The Art Reports*. Ed. by Rita Borgo. The Eurographics Association, 2015, pp. 105–125. DOI: 10.2312/eurovisstar.20151114.
- [5] Md Zahangir Alom et al. “The history began from alexnet: A comprehensive survey on deep learning approaches”. In: *arXiv preprint arXiv:1803.01164* (2018).
- [6] Jianwen Fu et al. “Malware visualization for fine-grained classification”. In: *IEEE Access* 6 (2018), pp. 14510–14523.
- [7] Sang Ni, Quan Qian, and Rui Zhang. “Malware identification using visualization images and deep learning”. In: *Computers & Security* 77 (2018), pp. 871–885.
- [8] Aminollah Khormali et al. “COPYCAT: practical adversarial attacks on visualization-based malware detection”. In: *arXiv preprint arXiv:1909.09735* (2019).
- [9] Daniel A Keim. “Information visualization and visual data mining”. In: *IEEE transactions on Visualization and Computer Graphics* 8.1 (2002), pp. 1–8.
- [10] André RA Grégio and Rafael DC Santos. “Visualization techniques for malware behavior analysis”. In: *Sensors, and Command, Control, Communications, and Intelligence (C3I) Technologies for Homeland Security and Homeland Defense X*. Vol. 8019. International Society for Optics and Photonics. 2011, p. 801905.

- [11] Lakshmanan Nataraj et al. “Malware images: visualization and automatic classification”. In: *Proceedings of the 8th international symposium on visualization for cyber security*. 2011, pp. 1–7.
- [12] Peter J Sackett et al. “A review of data visualization: opportunities in manufacturing sequence management”. In: *International Journal of Computer Integrated Manufacturing* 19.7 (2006), pp. 689–704.
- [13] RIDC NeuroMat. *Matriz de gráficos de dispersão*. Ed. by CC BY-SA 4.0. URL: <https://commons.wikimedia.org/w/index.php?curid=60578568>.
- [14] MathWorks. *Matrix of scatter plots by group*. URL: <https://www.mathworks.com/help/stats/gplotmatrix.html>.
- [15] M. Lanzenberger. *The Stardates - Visualizing Highly Structured Data*. Ed. by Proceedings of the Seventh International Conference on Information Visualization (IV’03). 2003.
- [16] Wikipedia. *Chernoff faces for evaluations of US judges*. 2010. URL: https://upload.wikimedia.org/wikipedia/commons/1/17/Chernoff_faces_for_evaluations_of_US_judges.svg.
- [17] Samuel Snider, Rafael Yuste, and Adam Packer. “A Neural Circuit for Angular Velocity Computation”. In: *Frontiers in neural circuits* 4 (Dec. 2010), p. 123. DOI: 10.3389/fncir.2010.00123.
- [18] Josh Saxe, David Mentis, and Chris Greamo. “Visualization of Shared System Call Sequence Relationships in Large Malware Corpora”. In: *Proceedings of the Ninth International Symposium on Visualization for Cyber Security. VizSec ’12*. Seattle, Washington, USA: Association for Computing Machinery, 2012, pp. 33–40. ISBN: 9781450314138. DOI: 10.1145/2379690.2379695. URL: <https://doi.org/10.1145/2379690.2379695>.
- [19] C. Lim et al. “Use of the circle segments visualization technique for neural network feature selection and analysis”. In: *Neurocomputing* 73 (2010), pp. 613–621.
- [20] Graham Oppy and David Dowe. “The turing test”. In: (2003).
- [21] Russell Stuart and Norvig Peter. *Artificial intelligence-a modern approach 3rd ed*. 2016.
- [22] Tom M Mitchell et al. “Machine learning”. In: (1997).
- [23] John R Koza et al. “Automated design of both the topology and sizing of analog electrical circuits using genetic programming”. In: *Artificial Intelligence in Design’96*. Springer, 1996, pp. 151–170.
- [24] Wikipedia. *Machine Learning*. 2020. URL: https://en.wikipedia.org/wiki/Machine_learning.

- [25] Jaime G Carbonell, Ryszard S Michalski, and Tom M Mitchell. “An overview of machine learning”. In: *Machine learning* (1983), pp. 3–23.
- [26] Li Deng and Dong Yu. “Deep learning: methods and applications”. In: *Foundations and trends in signal processing* 7.3–4 (2014), pp. 197–387.
- [27] George Cybenko. “Approximation by superpositions of a sigmoidal function”. In: *Mathematics of control, signals and systems* 2.4 (1989), pp. 303–314.
- [28] Leonardo Noriega. “Multilayer perceptron tutorial”. In: *School of Computing. Staffordshire University* (2005).
- [29] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. “Understanding of a convolutional neural network”. In: *2017 International Conference on Engineering and Technology (ICET)*. Ieee. 2017, pp. 1–6.
- [30] Aphex34. *Own work*. CC BY-SA 4.0.
- [31] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O’Reilly Media, 2019.
- [32] CR Rao and Venkat N Gudivada. *Computational analysis and understanding of natural languages: Principles, methods and applications*. Elsevier, 2018.
- [33] Benjamin Graham. “Fractional max-pooling”. In: *arXiv preprint arXiv:1412.6071* (2014).
- [34] Aphex34. *Own work*. CC BY-SA 4.0. URL: https://upload.wikimedia.org/wikipedia/commons/e/e9/Max_pooling.png.
- [35] Deepak Battini. *Implementing Drop Out Regularization in Neural Networks*. 2018. URL: <https://www.tech-quantum.com/implementing-drop-out-regularization-in-neural-networks/>.
- [36] Stack Overflow. *Binary file in Hex Editor*. 2017. URL: <https://i.stack.imgur.com/cVw1X.png>.
- [37] OpenClassRooms. *Aggrandir une image jusqu’aux pixels*. 2020. URL: <https://openclassrooms.com/forum/sujet/gtk-aggrandir-une-image-jusquaux-pixels>.
- [38] Xinbo Liu et al. “Atmpa: Attacking machine learning-based malware visualization detection methods via adversarial examples”. In: *2019 IEEE/ACM 27th International Symposium on Quality of Service (IWQoS)*. IEEE. 2019, pp. 1–10.
- [39] Manali Shaha and Meenakshi Pawar. “Transfer learning for image classification”. In: *2018 Second International Conference on Electronics, Communication and Aerospace Technology (ICECA)*. IEEE. 2018, pp. 656–660.

- [40] Mahbub Hussain, Jordan J Bird, and Diego R Faria. “A study on cnn transfer learning for image classification”. In: *UK Workshop on computational Intelligence*. Springer. 2018, pp. 191–202.
- [41] Isabelle Guyon et al. “A scaling law for the validation-set training-set size ratio”. In: *AT&T Bell Laboratories* 1.11 (1997).
- [42] Rosie Dunford, Quanrong Su, and Ekraj Tamang. “The pareto principle”. In: (2014).
- [43] Léon Bottou, Frank E Curtis, and Jorge Nocedal. “Optimization methods for large-scale machine learning”. In: *Siam Review* 60.2 (2018), pp. 223–311.
- [44] Diederik P Kingma and Jimmy Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).
- [45] Marcos Sebastián et al. “Avclass: A tool for massive malware labeling”. In: *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer. 2016, pp. 230–253.
- [46] Royi Ronen et al. “Microsoft malware classification challenge”. In: *arXiv preprint arXiv:1802.10135* (2018).
- [47] John Donahue, Anand Paturi, and Srinivas Mukkamala. “Visualization techniques for efficient malware detection”. In: *2013 IEEE International Conference on Intelligence and Security Informatics*. IEEE. 2013, pp. 289–291.
- [48] Ilya Loshchilov and Frank Hutter. “Sgdr: Stochastic gradient descent with warm restarts”. In: *arXiv preprint arXiv:1608.03983* (2016).
- [49] Chaofeng Wang, Zheng Li, and Jun Shi. “Lightweight image super-resolution with adaptive weighted learning network”. In: *arXiv preprint arXiv:1904.02358* (2019).
- [50] Sadaqat Ur Rehman et al. “Optimization of CNN through novel training strategy for visual classification problems”. In: *Entropy* 20.4 (2018), p. 290.

Appendix A

Malware visualizations by family

The following figures present the RGB tuned visualizations of one malware from each of the biggest families used in this work. For reasons of limited space, it is impossible to include malware from each of the 112 families with the 3 visualization techniques presented. That would be 336 images... The idea is to give a look at the diversity of the images obtained. The other visualization methods, grayscale and RGB, are not illustrated since RGB tuned is the most promising visualization method and to avoid cluttering up this work.

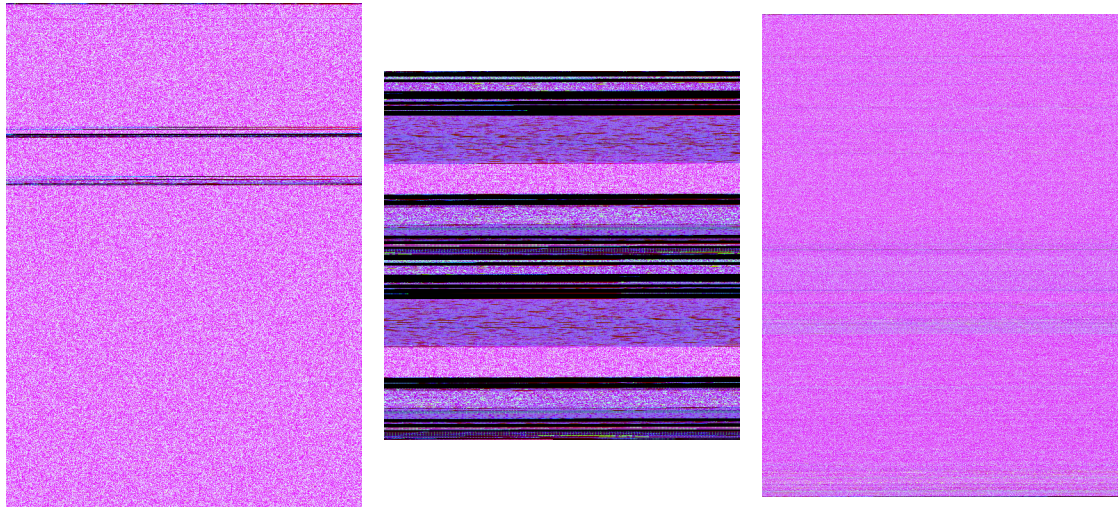


Figure A.1: Families benjamin (SHA256: 000c0b57 42e212bf 10c27e5f ea3f9e84), ausiv (SHA256: 000cc189 b87f2bb6 681ae769 66df65dc) and banload (SHA256: 000cde2d 5d5ba78f 189a3151 5d0e945e) (from left to right)

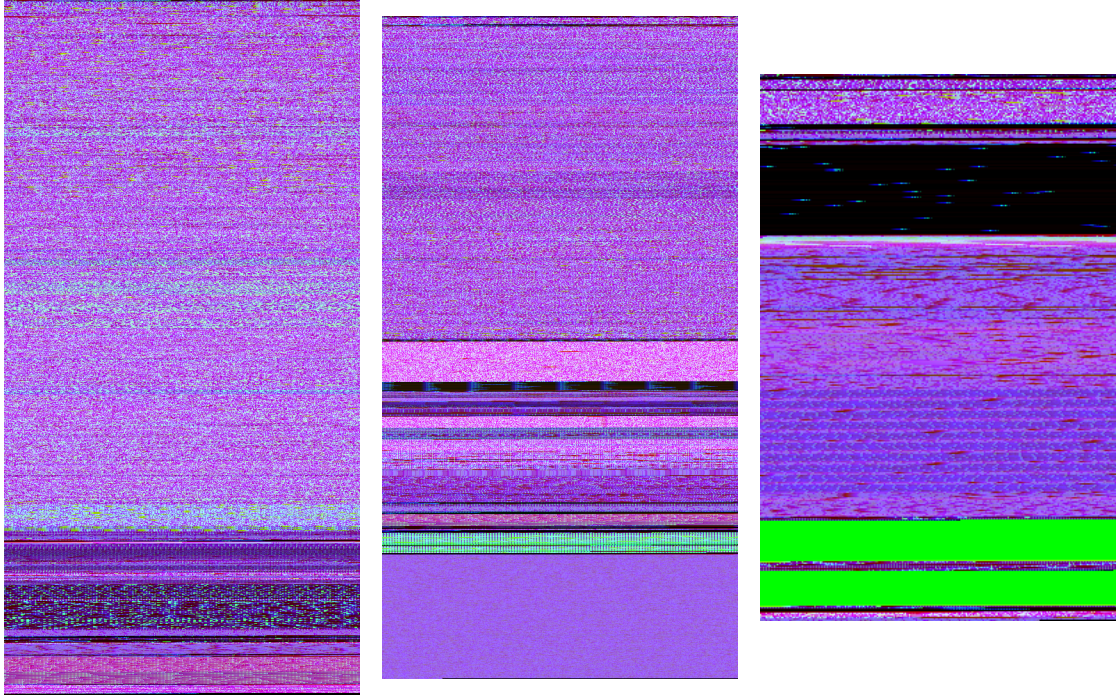


Figure A.2: Families browsefox (SHA256: 00a2a587 7bbc621f 1e8d7eaf 1f24e68a), drolnux (SHA256: 000af4fa 5083ad36 aaa2fd0a 52381954) and lamer (SHA256: 000c5408 cfc66c69 7285d1bf 89c87650) (from left to right)

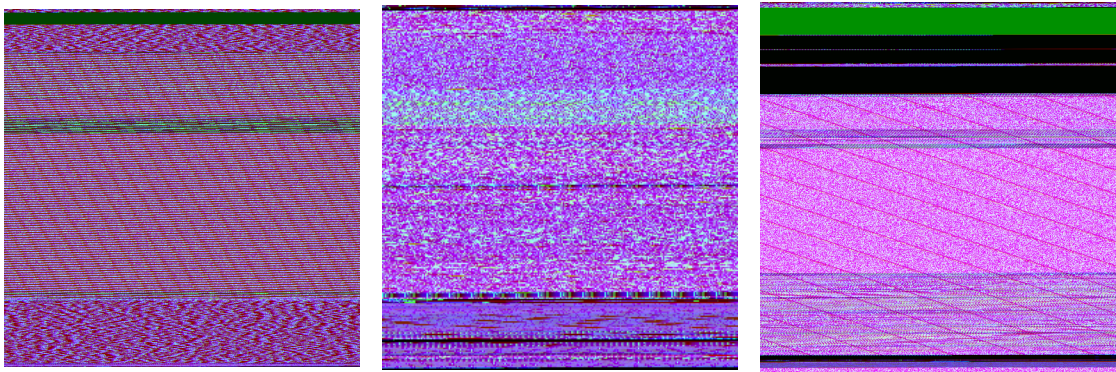


Figure A.3: Families qbot (SHA256: 00c6e021 e85df725 00baef46 590c993e), rozena (SHA256: 0073e91a 7b9d95dd 759b8fea 5dbe57bb) and shipup (SHA256: 000abb36 c450fec4 b89ea87d b7b91b14) (from left to right)

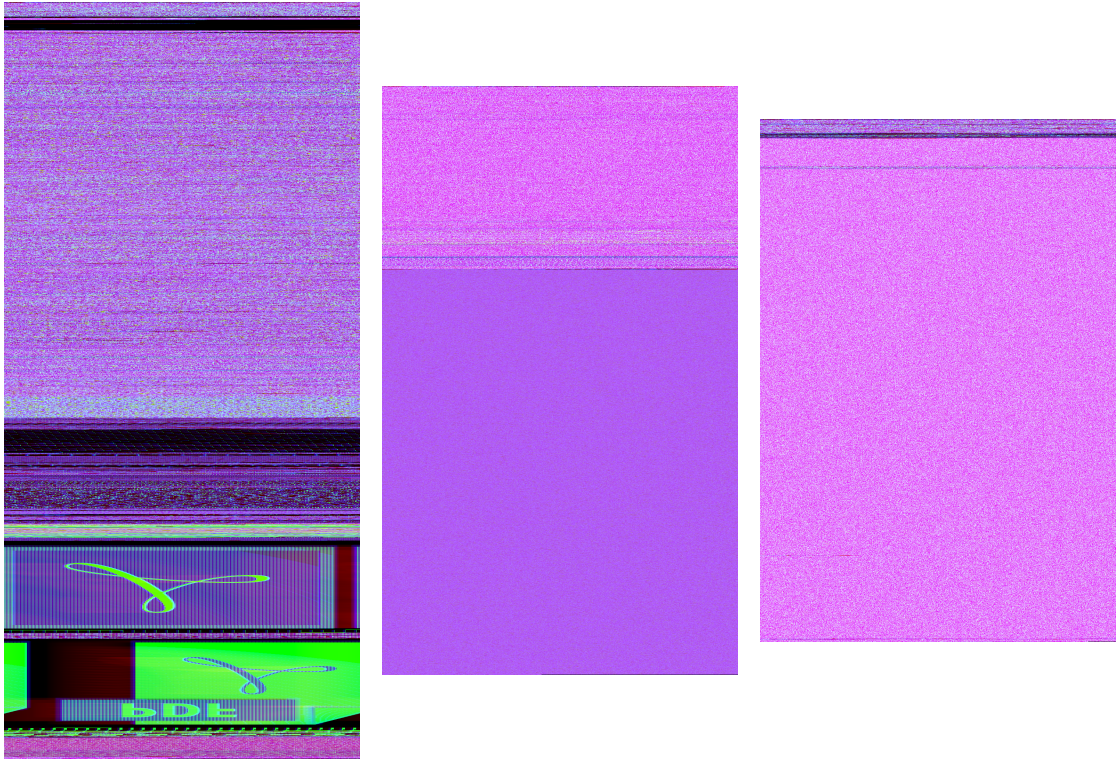


Figure A.4: Families gobot (SHA256: 011e978c 685d5a12 bda6c98c 73a2dbf3), high (SHA256: 0000b21d d4ac576d dba95156 7f2edf00) and installcore (SHA256: 000c6a48 e3bbd31a 23df66c3 9ddc854d) (from left to right)

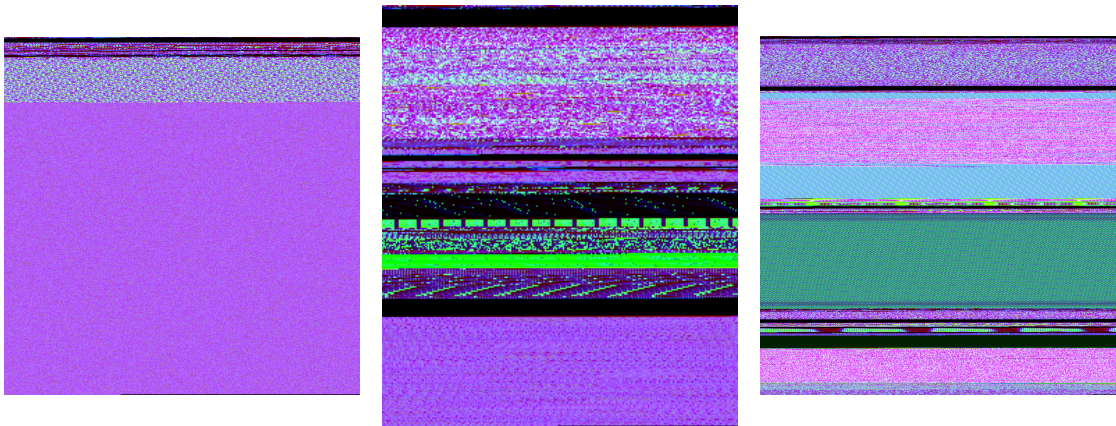


Figure A.5: Families mansabo (SHA256: 00a131e4 742b6d99 23e838e5 4cb00e22), nitol (SHA256: 000c95eb d4647282 a2cba525 6be4c1b0) and pioneer (SHA256: 00f2b2bf f0e04078 6d6ff394 57770bf9) (from left to right)

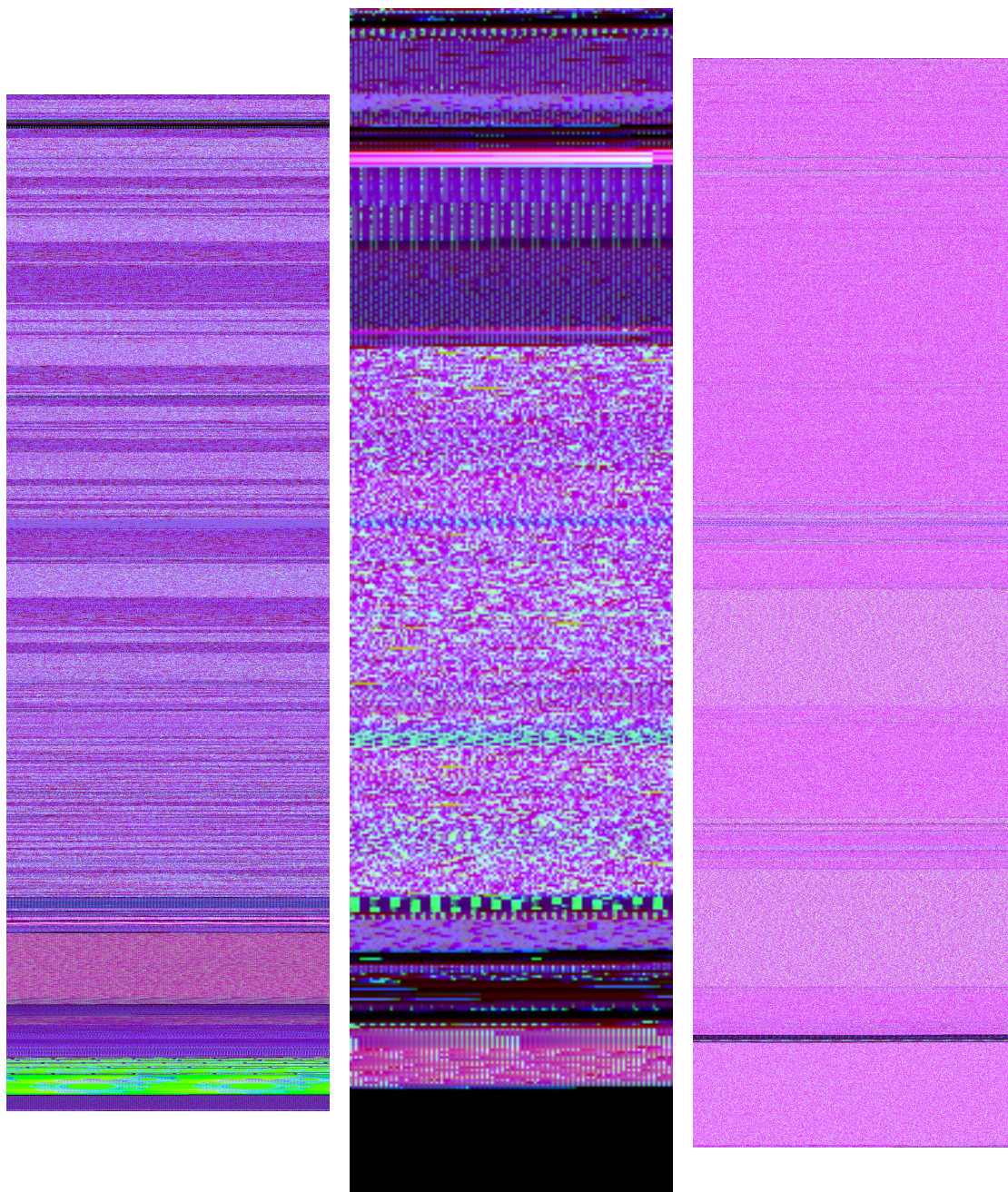


Figure A.6: Families banctean (SHA256: 000acd72 ac2628c8 733533dc a4afbb30), ceeinject (SHA256: 000c1c50 218d6a0e 5114670c 5e253273) and installmonster (SHA256: 000eab0e c8cb006d 6eb3a8c1 0b8a9b01) (from left to right)

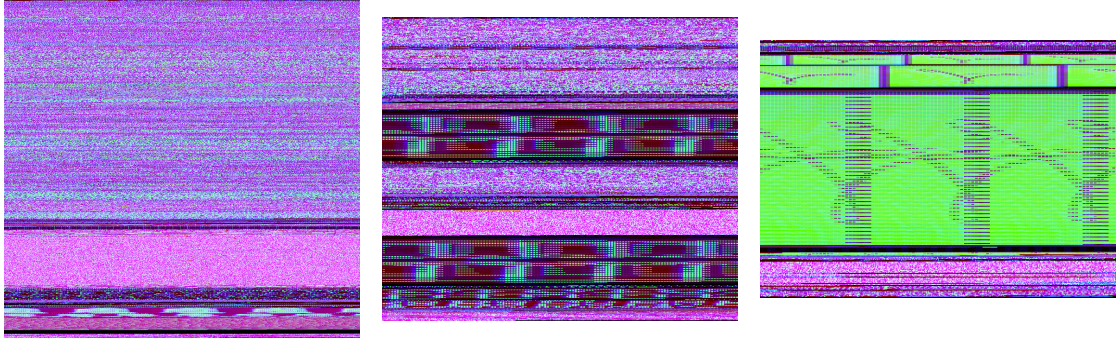


Figure A.7: Families downloadguide (SHA256: 0000f512 4830ef17 ee2091ab 45c502e3), delf (SHA256: 00a23cee b488befd fb90ad4e e42aba84) and wapomi (SHA256: 00b69bea 8f6611a4 1395932f 342bba9d) (from left to right)

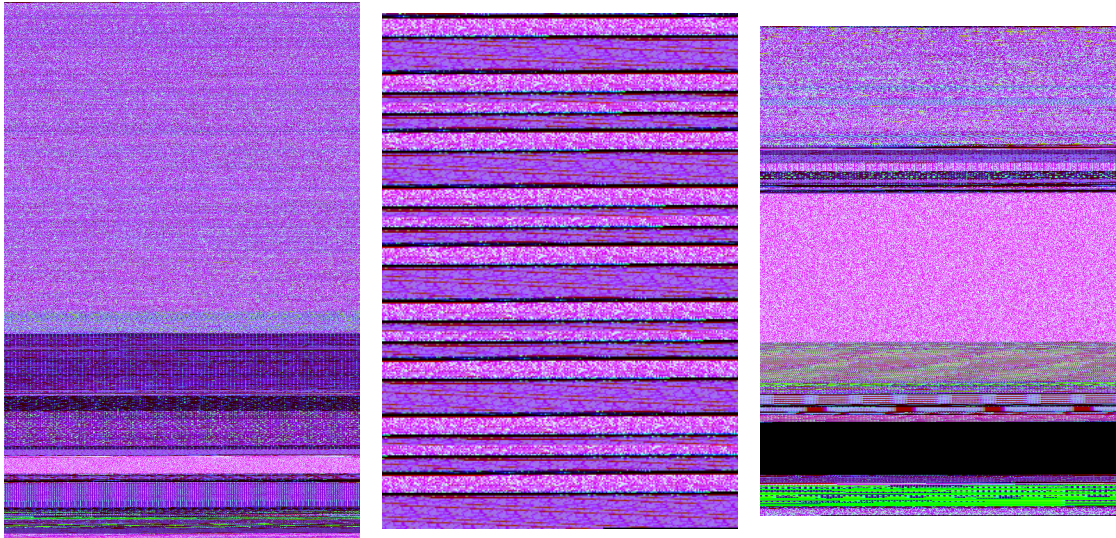


Figure A.8: Families cleanwares (SHA512: 00ad45fb 1aa57b6e 6f09be50 3ee2f355 78bbff87 b5da2bf3 87553422 e8f310e9), vtflooder (SHA256: 012a2fef 46cfa4df 64007ab9 932f7e6e) and gandcrab (SHA256: 00a7c8f3 f21a3d5f 769653e7 d922dcd7) (from left to right)

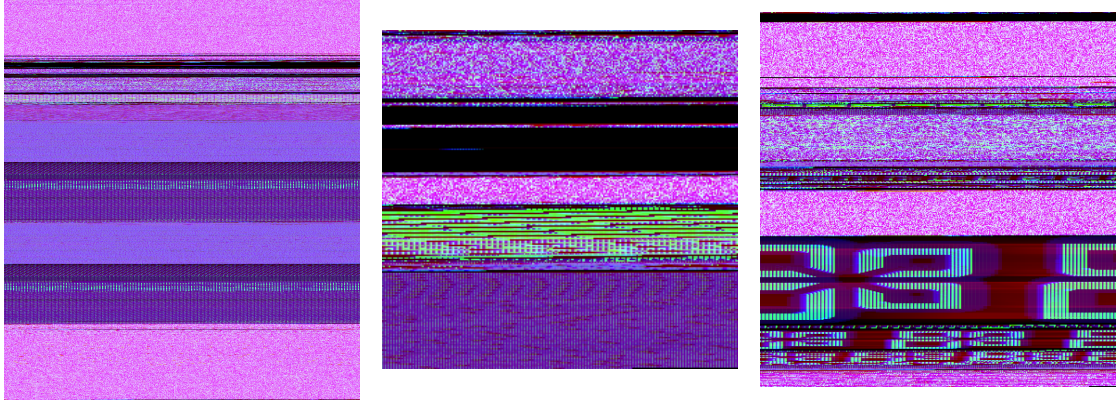


Figure A.9: Families stihat (SHA256: 000ca68a dacfdb1a 92eb533c e6c2f132), upatre (SHA256: 000c082b c55e68e4 51fd6ce3 a435b573) and ipamor (SHA256: 00b1b058 696b1afe 86405925 9ad2c9e7) (from left to right)

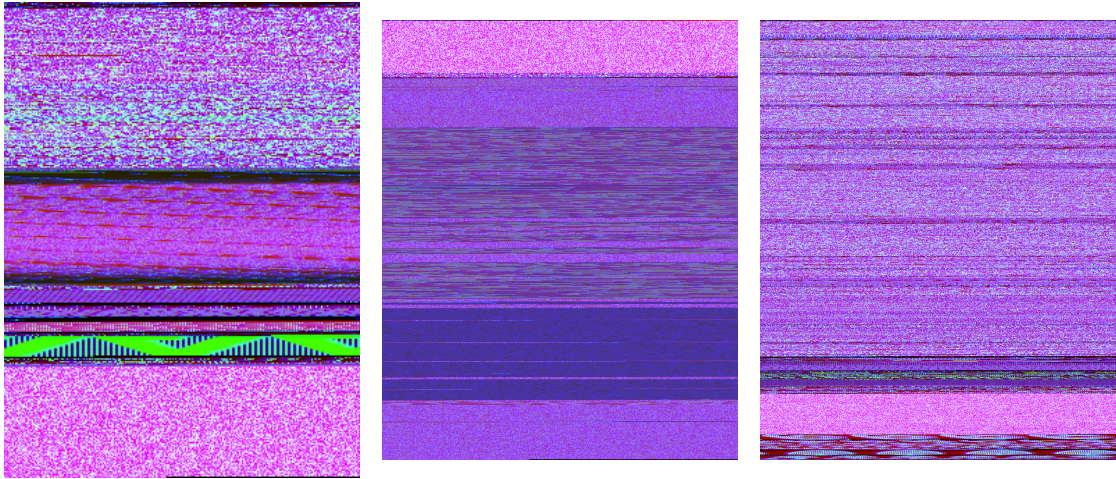


Figure A.10: Families wabot (SHA256: 000cecede 8032aff7 93e4de4f a42b41a6), dinwod (SHA256: 000ab4c9 a3f90e3f 8528857a 5c7dfb35) and zusy (SHA256: 000a8e50 c83cf1ad 32539e0e c37d05c0) (from left to right)

Appendix B

Presentation of the malware families

B.1 Cisco dataset

Table B.1 shows the distribution of the 13059 malwares in the 102 families of the dataset. We can observe the unbalance between the different families with the 'high' family alone constituted of 18.18% of all the dataset and the 'gobot' family with only 5 malwares and 0.04% of the dataset !

Family name	Number of malwares	Proportion (%)
1clickdownload	11	0.08
ainslot	14	0.11
allapple	33	0.25
amonetize	18	0.14
ausiv	71	0.54
autoit	319	2.44
banctean	77	0.59
banload	28	0.21
benjamin	124	0.95
berbew	753	5.77
blackmoon	87	0.67
blamon	40	0.31
browserfox	78	0.59
ceeinject	180	1.38
chir	38	0.29
cosmu	60	0.46
delf	527	4.04
dinwod	120	0.92
dlhelper	23	0.18
downloadassistant	15	0.11
downloadguide	130	0.99
drolnux	199	1.52

eggnog	38	0.29
emotet	35	0.27
fareit	56	0.43
filetour	19	0.15
firseria	51	0.39
flystudio	47	0.36
gandcrab	76	0.58
ganelp	48	0.37
gator	66	0.51
gepys	68	0.52
gify	28	0.21
gobot	5	0.04
hematite	65	0.49
high	2374	18.18
icloader	69	0.53
installbrain	18	0.14
installcore	157	1.20
installmonster	91	0.69
ipamor	148	1.13
ircbot	42	0.32
lamer	139	1.06
lebreath	36	0.28
lmir	11	0.08
loadmoney	45	0.34
lolbot	38	0.29
meaner	12	0.09
mansabo	193	1.48
memery	22	0.17
mepaow	83	0.64
mira	84	0.64
multiplug	35	0.27
mydoom	40	0.31
nitel	39	0.29
onlinegames	23	0.18
pepm	14	0.11
picsys	96	0.74
pioneer	24	0.18
poison	19	0.15
qbot	43	0.33
qqpass	36	0.28
ramnit	35	0.27
razy	324	2.48
rbot	127	0.97
reconyc	131	1.01
rozena	13	0.1
rubinurd	6	0.05
sfone	198	1.52
shakblades	15	0.11
shipup	70	0.54

shiz	95	0.73
shodi	92	0.70
shyape	46	0.35
sillyp2p	251	1.92
sillywnse	376	2.88
simbot	99	0.76
sivis	44	0.34
snojan	9	0.07
softcnapp	17	0.13
softpulse	47	0.36
stihat	381	2.92
sytro	121	0.93
tinba	14	0.11
tofsee	8	0.06
trickbot	9	0.07
ulpm	93	0.71
unruy	79	0.60
upantix	53	0.41
upatre	358	2.74
ursu	31	0.24
vbclone	88	0.67
virlock	248	1.89
virut	80	0.61
vtflooder	46	0.35
wabot	288	2.21
wacatac	35	0.27
wapomi	1119	8.57
widdit	8	0.06
xiaobaminer	22	0.17
zbot	43	0.33
zusy	479	3.67

Table B.1: Distribution of the malwares by family in the Cisco dataset

B.2 Kaggle dataset

The families of the Kaggle dataset are unknown. The families just have a numerous to distinguish them, from 1 to 9. Table B.2 shows the distribution of the 10868 malwares in the 9 families of the dataset.

Family name	Number of malwares	Proportion (%)
1	1541	14.18
2	2478	22.8
3	2942	27.07
4	475	4.37
5	42	0.39
6	751	6.91
7	398	3.66
8	1228	11.29
9	1013	9.32

Table B.2: Distribution of the malwares by family in the Kaggle dataset

B.3 Other datasets

The other datasets will not be discussed since the cleanwares are in on family of 668 samples and the custom dataset reuses 15 families (ausiv, autoit, bancteian, benjamin, blackmoon, browsefox, ceeinject, dinwod, downloadguide, drolnux, gandcrab, installcore, installmonster, ipamor and lamer) from the Cisco dataset.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN

École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl