arXiv:2206.05735v1 [cs.CR] 12 Jun 2022

FUSING FEATURE ENGINEERING AND DEEP LEARNING: A CASE STUDY FOR MALWARE CLASSIFICATION

Daniel Gibert, Quan Le
CeADAR
University College Dublin
Dublin, Ireland
{daniel.gibert, quan.le}@ucd.ie

Carles Mateu, Jordi Planes
Polytechnic School
University of Lleida
Lleida, Spain
{carles.mateu, jordi.planes}@udl.cat

ABSTRACT

Machine learning has become an appealing signature-less approach to detect and classify malware because of its ability to generalize to never-before-seen samples and to handle large volumes of data. While traditional feature-based approaches rely on the manual design of hand-crafted features based on experts' knowledge of the domain, deep learning approaches replace the manual feature engineering process by an underlying system, typically consisting of a neural network with multiple layers, that perform both feature learning and classification altogether. However, the combination of both approaches could substantially enhance detection systems. In this paper we present an hybrid approach to address the task of malware classification by fusing multiple types of features defined by experts and features learned through deep learning from raw data. In particular, our approach relies on deep learning to extract N-gram like features from the assembly language instructions and the bytes of malware, and texture patterns and shapelet-based features from malware's grayscale image representation and structural entropy, respectively. These deep features are later passed as input to a gradient boosting model that combines the deep features and the hand-crafted features using an early-fusion mechanism. The suitability of our approach has been evaluated on the Microsoft Malware Classification Challenge benchmark and results show that the proposed solution achieves state-of-the-art performance and outperforms gradient boosting and deep learning methods in the literature.

Keywords Malware Classification · Machine Learning · Deep Learning · Feature Extraction · Feature Fusion

1 Introduction

The fight against malware has never stopped since the dawn of computing. This fight has turned out to be a never-ending and cyclical arms race: as security analysts and researchers improve their defences, malware developers continue to innovate, find new infection vectors and enhance their obfuscation techniques. During the last decade, malware threats have been expanding vertically (i.e. numbers and volumes) and horizontally (i.e. types and functionality) due to the opportunities provided by technological advances [1]. According to AV-Test [2], the total number of new malware detections worldwide amounted to 677.66 million programs, and it is projected to surprass 700 million within 2020.

Traditionally, anti-malware engines relied on signature-based and heuristic-based methods to detect and block malware before they performed any damage. On the one hand, signature-based methods identify malware by comparing its code with the code of known malware that have already encountered, analyzed and recorded in a database. On the other hand, heuristic-based methods examine the code and behavior of malware to look for certain malicious behaviors and suspicious properties. Most anti-malware engines that employ heuristic analysis run the program commands within a specialized virtual machine, to isolate the suspicious program from the real machine. Although effective, this type of

[1] https://www.microsoft.com/en-us/security/business/security-intelligence-report
[2] https://www.av-test.org/en/statistics/malware/

https://t.me/learningnets

analysis is very time consuming because it involves setting up a safe environment every time a suspicious file has to be analyzed.

As a result, the diversity, sophistication and availability of malicious software make the task of securing computer networks and systems very challenging, and force security analysts and researchers to constantly improve their cyberdefenses to keep pace with the attacks. During the last decade, due to the massive growth of malware streams, organizations faced the daunting challenge of dealing with thousands of attacks a day while also experiencing a shortage of cybersecurity skills and talent [1]. In consequence, new methods started to be adopted to complement traditional detection approaches and keep pace with new attacks and variants. This scenario presented a unique opportunity for machine learning, as an alternative to signature-based approaches, to impact and revamp the cybersecurity landscape, because of its ability to generalize to never-before-seen malware and to handle large volumes of data.

Traditional machine learning approaches for malware detection and classification rely on the manual extraction of hand-crafted features defined by experts in the domain [2, 3, 4, 5, 6]. However, these solutions depend almost entirely on the ability of the experts to extract characterizing features that accurately represent malware, and, the computational and memory requirements needed to extract some of these discriminant features, such as N-grams, limit their applicability in the real-world [7]. Lately, various approaches have been presented to automatically learn to extract N-gram like features from malware without having to exhaustively enumerate all N-grams during training using deep learning [8, 9], and approaches to automatically extract features from malware's structural entropy representation [10], gray-scale image representation [11, 12], from the binary code or any compressed representation of it [13, 14]. For a complete review of feature-based and deep learning approaches we refer the readers to [15] and the references therein.

The task of malware detection and classification includes multiple types of features and thus, by only taking as input the raw bytes or opcodes sequence a great deal of useful information for classification is ignored such as the characteristics of the Portable Executable (PE) Headers, the import address table (IAT), etc. As a result, deep learning approaches tend to perform poorly in comparison to multimodal approaches in the literature [2, 3].

This paper presents an hybrid approach for malware classification that addresses the aforementioned limitation of deep learning approaches by combining deep and hand-crafted features using simple, but yet effective, early fusion mechanism to train gradient boosting trees [16]. As far as we know, this is the first approach to ever try to combine hand-crafted features defined by experts with features automatically learned through deep learning for the task of malware detection and classification. The main idea behind is to fuse both the deep and hand-crafted features into a single feature vector that is later used to train a single model to learn the correlation and interactions between each type of features. Our approach extracts well-known features such as API function calls, section characteristics, the frequency of usage of the registers, entropy features, and so on, and also extracts N-gram like features from the binary content and the assembly language source code of malware [8, 9], plus deep features from malware's structural entropy [10] and gray-scale image representation [11]. This approach has been extensively assessed on the dataset provided by Microsoft for the Big Data Innovation Gathering Challenge of 2015 [17], which has become the de facto standard benchmark to evaluate machine learning models for malware classification. Results show a 99.81% 10-fold cross validation accuracy on the training set and a 0.0040 logarithmic loss on the test set, outperforming any feature-based and deep learning-based approach in the literature.

The rest of the paper is organized as follows. Section 2 describes the methods employed to detect and classify malicious software. Section 3 introduces the related research to address the problem of malware detection and classification. Section 4 provides a detailed description of our system, and the types of features extracted. Lastly, Section 5 describes the experimentation and Section 6 summarizes our research and presents some remarks on the ongoing research trends.

2 Background

Next, it is introduced the necessary background required for the reader to understand malware forensics and the recently rise of machine learning to complement traditional detection methods.

2.1 The Task of Malware Detection and Classification

Malicious software, also known as malware, is any kind of software that has been specifically designed to disrupt, harm or exploit any computer system or network. Typically, malware with similar characteristics and common behavior are grouped into families, whereas a malware family usually encompasses samples of malware that have been generated from the same code base. As malware keeps evolving, new variants or strains of a family might arise showing similar traits in their variations, just as these families have. These variations usually differ in key areas, such as those dealing with payload and infection. Therefore, the task of malware detection refers to the task of detecting whether a given piece of software is malicious or not while the task of malware classification refers to the task of distinguishing and

https://t.me/learningnets

classifying malware into families. Although the task of detecting malware is crucial to stop and prevent an attack before it causes damage, the task of malware classification helps to better understand how the malicious software has infected the system, its threat level and potential damage, and what can be done to defend against it and recover from the attack.

2.2 Traditional Detection Techniques

Traditionally, to detect malware, anti-malware engines relied on signature and heuristic detection. On the one hand, signature detection involves comparing a program's code with the code of known malware that has already been encountered, analyzed and stored in a database, in order to find footprints matching those of known malware. Typically, if a program's code match one or more of those footprints, it is labelled as malicious and it will be either deleted or put into quarantine. For decades, this method of identifying malware has been the primary technique employed by anti-malware engines because of its simplicity. However, this detection technique solely works against known malware, limiting its protection against new forms of malware. In addition, modern malware can alter its signature to avoid detection by employing code obfuscation, dynamic code loading, encryption, packing, etc.

To counter the limitations of signature detection, anti-malware engines adopted heuristic detection. Oppositely to signature detection, which looks to match signatures or footprints found in known malicious files, heuristic detection uses rules and/or algorithms to look for pieces of code which may indicate malicious intent. For instance, a common technique is to look for specific commands or instructions that are not typically found in a benign application such as the payload of some trojan, the replication mechanism of virus, etc. Traditional heuristic anti-malware engines use some kind of rule-based or weight-based system to determine the maliciousness or the threat that a program poses to the computer system or network. If these rules exceed a predetermined threshold, then an alarm is triggered and a precautionary action is taken. However, these rules and heuristics have to be previously defined by experts after analyzing the behavior of malware, which is a complex and time consuming process, even for security experts.

A few decades ago, the number of threats attributed to malicious software was relatively low and simple hand-crafted rules were sufficient to detect the ongoing threats. However, during the last decade, malware has exploded in terms of diversity, sophistication, and availability, and thus, the aforementioned detection and analysis techniques have been unable to keep pace with the new attacks and variants. In addition, the shortage of experienced security researchers and analysts [1] plus the complexity of forensic investigations have contributed to strengthening the use of machine learning as an appealing signature-less alternative to detect malware because of (1) its ability to generalize to never-before-seen malware and (2) to handle large volumes of data.

3 Related Work

In this section, the related studies performed in the field of malware detection and classification that are powered by machine learning are presented. For a complete review of the features and deep learning architectures defined by experts to build machine learning detection systems we refer the readers to Ucci et al. [18], Gibert et al. [15], and the references therein.

Traditional machine learning approaches for malware detection and classification rely on the manual extraction of features defined by security experts [2, 3, 4, 5, 6]. Feature extraction is one key step to build a malware detection and classification system. It transforms raw data, i.e. binary executables, into numerical features that provide an abstract representation of the original data. Common features are byte and opcode N-grams, API function calls, the frequency of use of registers, characteristics extracted from the header and sections of executables, etcetera. However, these solutions depend almost entirely on the ability of the security experts to extract a set of descriptive features that accurately represent the malware characteristics. In addition, the computational and memory requirements needed to extract some of these features, such as N-grams, far exceeds the capabilities of most existing systems and limit their applicability in the real-world [7].

Lately, various approaches have been presented to automatically learn to extract N-gram like features from malware without having to exhaustively enumerate all N-grams during training through deep learning. For instance,Gibert et al. [8, 9] proposed a shallow convolutional neural network architecture that intrinsically learn to extract N-gram like features from both the malware's binary content and its assembly language source code, respectively. This is achieved by a convolutional layer with filters of various sizes followed by a global max-pooling layer, which allow the model to retrieve the features regardless of their location in the assembly language instructions and byte sequences. Recently, other approaches [19, 20] have been proposed to detect malware based on their binary content. Raff et al. [19] proposed a shallow convolutional neural network architecture with a gated convolutional layer to capture the high location invariance in malware's binary content while Krčál et al. [20] presented the deepest architecture to date, consisting of

https://t.me/learningnets

an embedding layer followed by two convolutions, one max-pooling layer, two more convolutions, and a global average pooling layer and four fully connected layers.

However, dealing with raw byte sequences directly is very challenging as it involves the classification of sequences of extreme lengths. For instance, the size of the executables may range from some KBs to hundreds of MBs, e.g. 100 MB correspond to a sequence of 100,000,000 bytes. To deal with these sequences of extreme length, various approaches [14, 10, 11, 13, 12, 21, 22, 23, 24] proposed to compress the information in the malware's binary content. For instance, Gibert et al. [10] presented an approach to classify malware represented as a stream of entropy values. In their work, the binary content is divided into chunks of code of fixed size and afterwards, the information at each chunk is compressed by calculating its entropy value. On the other hand, executables could be represented as grayscale images [11, 12, 23, 24]. To represent a malware sample as a grayscale image, every byte has to be interpreted as one pixel in an image, where values are in the [0,255] (0:black, 255:white). Afterwards, the resulting 1-D array has to be reorganized as a 2-D array.

Nevertheless, the task of malware detection and classification is regarded as a multimodal task, as it includes multiple types of features and thus, by only taking as input the assembly language instructions and the raw bytes sequence or a compressed representation of it, a lot of useful information for characterizing malware is ignored. As a result, these unimodal deep learning approaches that only take as input a single source of information tend to perform poorly in comparison to multimodal approaches [2, 3, 25, 26] that extract different types of features from various modalities of information. This is because these types of features or modalities provide complementary information to each other, and reflect patterns not visible when working with individual modalities on their own. In fact, these approaches that extract multiple types of features from malware remained unbeaten in terms of classification performance and have been the way to go for detecting and classifying malware.

To solve the aforementioned issues of deep learning approaches, we present a multimodal system that combines the benefits of feature engineering and deep learning to achieve state-of-the-art results in the task of malware detection and classification.

4 System Overview

The proposed classification system extracts hand-engineered and deep features to address the task of malware classification by combining the aforementioned features using a simple, but yet effective, early fusion mechanism in order to train gradient boosting trees [16] . Following, the components of the proposed system are described in detail. First, the different types of features extracted are explained. Then, the deep learning architectures implemented to automatically learn and extract deep features from raw data are introduced. Afterwards, it is described the proposed fusion mechanism to combine the aforementioned features. Lastly, it is introduced the machine learning algorithm trained to categorize malware into families. See Figure 1 for an overview of the system.

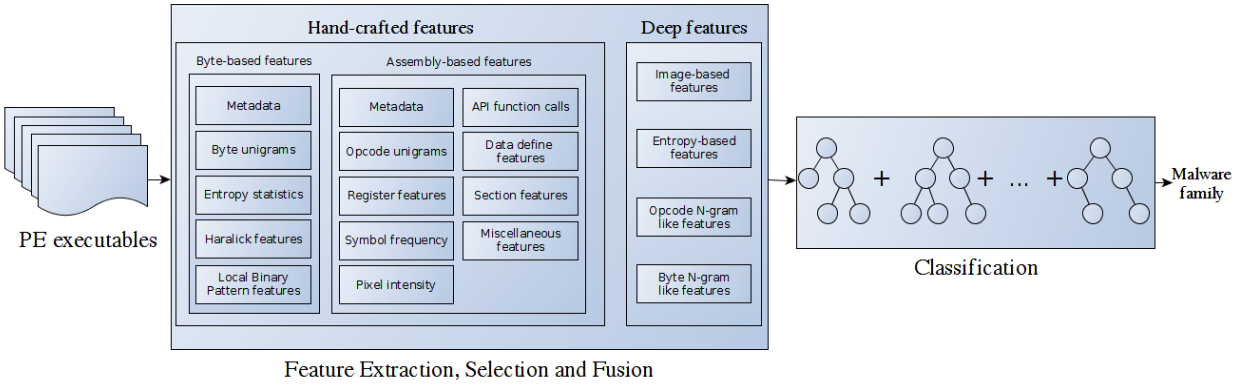


Figure 1: Overview of the proposed malware classification system.

4.1 Hand-engineered Features

This system has been implemented to tackle the problem of Windows malware classification and thus, the features have been specifically designed to be extracted from executables in the Portable Executable (PE) file format, which is the format for executables, DLLs, FON Font files and others in the Windows operating system.

https://t.me/learningnets

There are two common representations of a Windows malware executable: (1) the hexadecimal representation of malware's binary content (or hex view) and (2) the assembly language source code of malware (or assembly view). On the one hand, the hex view of malware represents the machine code as a sequence of hexadecimal values. Cf. Figure 2. The first value indicates the starting address of the machine codes in the memory, and each hexadecimal value

```
00401000 E8 0B 00 00 00 E9 16 00 00 00 90 90 90 90 90 90
00401010 B9 C5 3C 57 00 FF 25 60 E2 40 00 90 90 90 90 90
00401020 68 30 10 40 00 E8 D4 B2 00 00 59 C3 90 90 90 90
00401030 B9 C5 3C 57 00 FF 25 5C E2 40 00 90 90 90 90 90
00401040 E8 0B 00 00 00 E9 16 00 00 00 90 90 90 90 90 90
00401050 B9 C4 3C 57 00 FF 25 54 E2 40 00 90 90 90 90 90
00401060 68 70 10 40 00 E8 94 B2 00 00 59 C3 90 90 90 90
00401070 B9 C4 3C 57 00 FF 25 58 E2 40 00 90 90 90 90 90
00401080 8B 0D 10 2E 57 00 B8 12 00 00 00 3B C8 7D 05 A3
00401090 74 2D 57 00 83 3D 04 2E 57 00 5B 7E 18 8B 0D CC
004010A0 2D 57 00 B8 04 00 00 00 2B C8 89 0D CC 2D 57 00
004010B0 33 C0 C2 08 00 A1 48 2E 57 00 2D D9 00 00 00 74
004010C0 06 29 05 CC 2D 57 00 33 C0 C2 08 00 90 90 90 90
004010D0 A1 CC 2D 57 00 A3 80 2D 57 00 83 C0 1D 75 0A 81
004010E0 05 10 2E 57 00 D7 00 00 00 B8 3C F0 40 00 C2 04
004010F0 00 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
00401100 A1 F0 2D 57 00 8B 15 58 2E 57 00 83 EC 08 3B D0
00401110 B9 18 00 00 00 7E 06 29 0D 00 2E 57 00 53 56 50
00401120 89 0D 64 2E 57 00 E8 D5 11 00 00 88 44 24 0C E8
00401130 2C 53 00 00 E8 97 73 00 00 8B D8 83 3D 80 2D 57
00401140 00 61 0F 8D 41 01 00 00 A1 F0 2D 57 00 55 BD 25
00401150 00 00 00 83 E3 1E 3B C5 57 C6 44 24 10 DD 7E 6D
00401160 A1 48 2E 57 00 8B 35 58 2E 57 00 50 68 50 F0 40
00401170 00 68 50 F0 40 00 BF 23 00 00 00 83 F6 36 E8 6D
```

Figure 2: Hexadecimal view of malware.

(byte) carry meaningful information of the Portable Executable file such as instruction codes and data. On the other hand, the assembly language source code contains the symbolic machine code of the executable, i.e. the machine code instructions, as well as function calls, memory allocation and variable information. Cf. Figure 3. To obtain the source code, malware binaries have to be first disassembled. Typically, it is used the Interactive Disassembler (IDA) tool [3], Radare2 [4] or Ghydra [5], among others. To build an accurate classifier, we propose to extract hand-crafted features from

```
.text:00470580                  loc_470580: ; CODE XREF: start+43□j
.text:00470580                              ; start+5A□j ...
.text:00470580 57                   push    edi
.text:00470581 BB A7 81 AA A4       mov     ebx, 0A4AA81A7h
.text:00470586 09 DB                or      ebx, ebx
.text:00470588 74 12                jz      short loc_47059C
.text:0047058A 81 FB A9 D1 D0 1E    cmp     ebx, 1ED0D1A9h
.text:00470590 75 0A                jnz     short loc_47059C
.text:00470592 85 DB                test    ebx, ebx
.text:00470594 75 06                jnz     short loc_47059C
.text:00470596 89 9D 4C FF FF FF    mov     [ebp+var_B4], ebx
.text:0047059C
.text:0047059C                  loc_47059C: ; CODE XREF: start+8C□j
.text:0047059C                              ; start+94□j ...
.text:0047059C 8B 0D 04 10 4B 00    mov     ecx, dword_4B1004
.text:004705A2 8B 85 3C FF FF FF    mov     eax, [ebp+var_C4]
.text:004705A8 89 85 38 FF FF FF    mov     [ebp+var_C8], eax
.text:004705AE 83 F9 15             cmp     ecx, 15h
.text:004705B1 75 0B                jnz     short loc_4705BE
.text:004705B3 81 F9 42 37 2E 74    cmp     ecx, 742E3742h
.text:004705B9 74 03                jz      short loc_4705BE
.text:004705BB 89 4D A8             mov     [ebp+var_58], ecx
```

Figure 3: Assembly view of malware.

both representations, the hex view and the assembly view of the executables, to exploit the complementary information provided by these two representations. More specifically, we decided to limit our proposed approach to only common and well-known features [2, 3] in order to evaluate whether or not the classification performance of the system improves after the addition of deep features extracted by the deep learning models. Next, the hand-crafted features extracted from the hexadecimal source code of malware's binary content and the assembly language source code are presented.

[3] https://www.hex-rays.com/products/ida/

[4] https://rada.re/n/

[5] https://www.nsa.gov/resources/everyone/ghidra/

https://t.me/learningnets

4.1.1 Hexadecimal-based features

Hexadecimal-based features refer to those features extracted from the hexadecimal representation of malware's binary content. Following you will find a brief description of all the hexadecimal-based feature types used by our system.

Metadata information (*BYTE_MD*). Two features compose the metadata information extracted from the hexadecimal view of malware, (1) the size of the file and (2) the address of the first byte sequence.

Byte unigram features (*BYTE_1G*). Remember that in a byte sequence, each element can take one out of 256 different values, i.e. ranging from 0 to 255 (the byte range). In addition, some elements are represented by the special symbol ??, indicating that the corresponding byte has no mapping in the executable file (the contents of those addresses are uninitialized within the file). However, better results can be achieved by only taking as input the 256 byte values [2]. For this reason, we decided to leave the ?? out the experiments.

Entropy-based features (*BYTE_ENT*). Entropy has long been used in the security industry to detect the presence of encrypted and compressed code as these tend to have higher entropy than native code [27]. To sum up, the entropy of a byte sequence refers to the amount of disorder of the distribution of bytes, whose value range from 0 (order) to 8 (randomness). Generally speaking, if occurrences of all byte values are the same, the entropy will be larger, but if certain byte values occur with higher probability, the entropy will be smaller. Nevertheless, the use of simple entropy statistics may not be enough to detect malware as authors usually try to conceal the encrypted and compressed code in a way that they bypass high entropy filters. For this reason, the bytes sequence is typically represented as a stream of entropy values (aka structural entropy). See Figure 4. To calculate the structural entropy of an executable, the bytes sequence is split into non-overlapping chunks of fixed size, e.g. 256, and for each chunk of code, its entropy value is computed as follows:

$$H(X) = -\sum_{i=1}^{n} p(i) \cdot log_b p(i) \tag{1}$$

where $H(X)$ is the measured entropy value of a given chunk of code X with values $x_1, \ldots, x_j$, j is the number of values in X, $p(i)$ refers to the probability of appearances of the byte value i in X and n is equal to 255, i.e. byte code values are in the range of [0, 255].

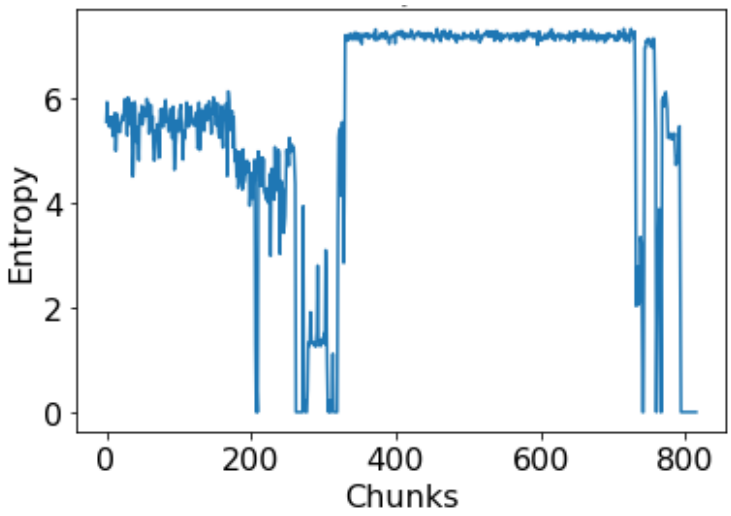


Figure 4: Structural entropy of a malware sample.

Consequently, we extracted various statistical features from the structural entropy of executables such as the mean, variance, median, maximum, minimum entropy values, and the percentiles.

IMG-based features (*BYTE_HARALICK*, *BYTE_LBP*). Nataraj et al. [28] proposed a method for visualizing and classifying malware using image processing techniques. In their work, malware binaries are visualized as grayscale images, with every byte reinterpreted as one pixel in the image. Then, the resulting array is reorganized as a 2-D array and visualized as a grayscale image, where values are in the range [0, 255], 0 for black and 255 for white. The rational behind this representation is that images of malicious software from a given family are similar between them but distinct from those belonging to a different family and thus, this visual similarity can be exploited to classify malware. Cf. Figure 5.

https://t.me/learningnets

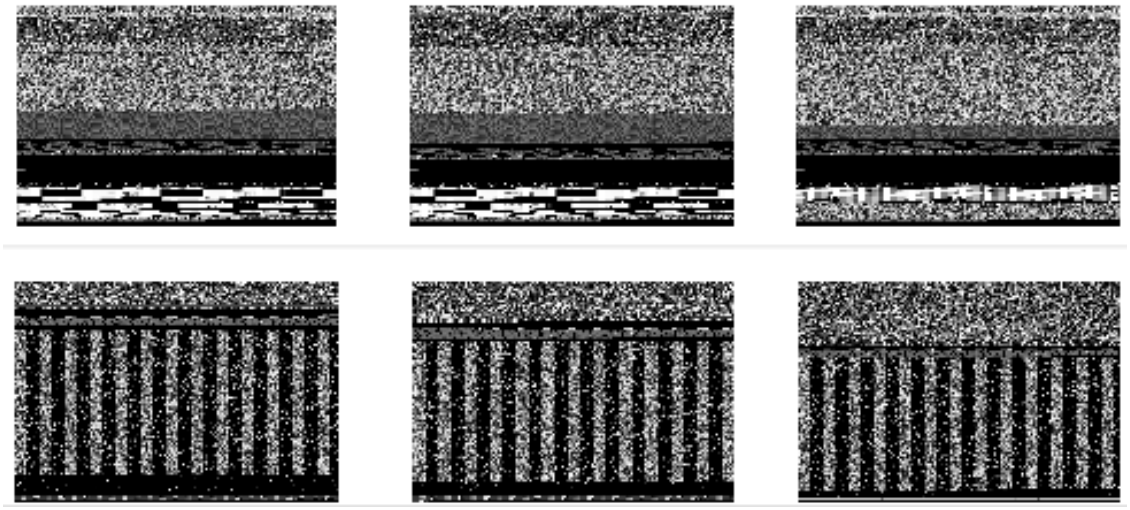


Figure 5: Grayscale image representation of samples belonging to the Obfuscator.ACY and Gatak families, respectively. Notice that the images of malicious software belonging to the same family are similar between them while distinct from those images of malware belonging to the other family.

From these visual representation, we extracted Haralick [29] and Local Binary Pattern [30] features. On the one hand, Haralick features are calculated by constructing a co-occurence matrix and computing the Haralick equations, e.g. angular second-moment, contrast, correlation, etc. On the other hand, the local binary pattern features of a given pixel are computed as follows. First, a 8 bit binary array is initialized with all values equal to 0. Then, the given pixel is compared with its neighboring pixels in clockwise direction. If the value of the neighboring pixel is greater or equal to 1 is assigned to its corresponding position. This results in a 8 bit binary array with zeros and ones. Following, the 8-bit binary pattern is converted to a decimal number and it is stored in the corresponding pixel location in the LBP mask. Finally, this process is applied to all pixels in the image and, once the LBP values of all pixels have been calculated, the mask is normalized, resulting in 256 features.

4.1.2 Assembly-based features

Assembly-based features refer to those features extracted from the assembly language source code of malware. Following you will find a brief description of the the assembly-based feature types used by our system.

Metadata information (*ASM_MD*). Two features compose the metadata information extracted from the assembly language source code of malware, (1) the size of the file and (2) the number of lines in the file.

Operation codes unigram features (*ASM_OPC*). An operation code or opcode is the portion of a machine language instruction that specifies the operation to be performed, e.g. *ADD*, *MUL*, *SUB*, etc. Instead of calculating the frequency of all opcodes, we just selected a subset of 93 opcodes based on their commonness and frequency of use in malicious applications [31]. The full list of opcodes extracted from each assembly language source code file can be found in the project's repository [6].

Data define features (*ASM_DD*). Data define directives are used for reserving storage for variables. However, because of packing, some malware samples do not contain any API call and barely a few operation codes. More specifically, those samples of malware mostly contain db, dw and dd define directives, which are used for setting byte, word, and double words, respectively. Subsequently, the frequency of use of the aforementioned data define directives has high discriminative power for a number of malware families. The full list of data define features can be found in Table 1.

Register features (*ASM_REG*). Registers are like variables built in the processor. Using registers instead of memory to store values makes the process faster and cleaner. The x86 architecture of processors contains eight General-Purpose Registers. All registers can be accessed in 16-bit and 32-bit modes. In 16-mode, the registers are abbreviated using the abbreviations listed in Table 2. In 32-bit mode, it is added the prefix 'E' to this two-letter abbreviation. For instance, 'ECX' it the counter register as a 32-bit value. Similarly, in the 64-bit version, the 'E' is replaced with an 'R'. Thus, the 64-bit version of 'ECX' is 'RCX'. In addition, the first four registers,

[6]

https://t.me/learningnets

Table 1: List of features in the *ASM_DD* category [2].

Feature Name	Description
db_por	Proportion of the db data define directive to the whole file
dd_por	Proportion of the dd data define directive to the whole file
dw_por	Proportion of the dw data define directive to the whole file
dc_por	Proportion of the db, dd and dw data define directives to the whole file
db0_por	Proportion of the db data define directive with 0 parameters to the whole file
dbN0_por	Proportion of the db data define directive with more than 0 parameters to the whole file
db_text	Proportion of the db data define directive in the .text section
db3_rdata	Proportion of the db data define directive with one non-zero parameter to the .rdata section
db3_data	Proportion of the db data define directive with one non-zero parameter to the .data section
db3_idata	Proportion of the db data define directive with one non-zero parameter to the .idata section
db3_all	Proportion of the db data define directive with one non-zero parameter to the whole file
db3_NdNt	Proportion of the db data define directive with one non-zero parameter in the unknown sections
db3_zero_all	Proportion of the db data define directive with 0 parameter with respect to the number of db data define directives with non-zero parameters
dd_text	Proportion of the dd data define directive in the text section
dd_rdata	Proportion of the dd data define directive to the .rdata section
dd4	Proportion of the dd data define directive with four parameters
dd5	Proportion of the dd data define directive with five parameters
dd6	Proportion of the dd data define directive with six parameters
dd4_all	Proportion of the dd data define directive with four parameters to the whole file
dd5_all	Proportion of the dd data define directive with five parameters to the whole file
dd6_all	Proportion of the dd data define directive with six parameters to the whole file
dd4_NdNt	Proportion of the dd data define directive with four parameters in unknown sections
dd5_NdNt	Proportion of the dd data define directive with five parameters in unknown sections
dd6_NdNt	Proportion of the dd data define directive with six parameters in unknown sections

Table 2: 16-bit naming convention of the eight General-Purpose Registers (GPR).

Register type	Description
Accumulator register (AX)	Employed in arithmetic operations (e.g. INC, DEC, ADD, SUB, etc).
Counter register (CX)	Employed in shift/rotate instructions and loops (e.g. JMP, JNZ, etc)
Data register (DX)	Employed in arithmetic and I/O operations (e.g. IN, INS, OUT, etc)
Base register (BX)	Employed as a pointer to data
Stack Pointer register (SP)	Pointer to the top of the stack
Stack Base Pointer register (BP)	Pointer to the base of the stack
Source Index register (SI)	Pointer to the source in stream operations
Destination Index register (DI)	Pointer to the destination in stream operations

AX, CX, DX and BX, can be addressed as two 8-bit halves [7]. Moreover, there are six Segment registers used to store the starting addresses of the code, the data and the stack segments, namely the Stack Segment, the Code Segment, the Data Segment, the Extra Segment, the F Segment and the G Segment.

Symbols frequency (*ASM_SYM*). The usage of the following set of symbols -, +, *,], [, ?, @ are extracted as features because these characters are typically found in code that has been designed to evade detection by resorting to indirect calls and to dynamic library loading.

Application Programming Interface (ASM_API). Application Programming Interface (API) functions and system calls are related to services provided by the operating systems, in our case Windows. These functions provide access to key operations such as network, security, system services, file management, and so on, and there are the only way for software to access system resources managed by the operating system. As a result, API function calls provide descriptive information with respect to the intent or behavior of a particular piece of software. For this reason, we measured the frequency of use of a subset of 794 API functions based on their frequency on a study of nearly 500 thousand malicious samples [32]. The rational behind using only a subset of API functions is that the total number of APIs is extremely large and considering all of them would bring little to no meaningful information to the task of malware classification. The subset of API functions extracted from the assembly language source code file can be found in the project's repository [8].

Pixel intensity features (*ASM_PIXEL*). Just as malware's binary content, the assembly language source code of malware can also be visualized as a grayscale image. Cf. Figure 6. In fact, the Winner's of the Big Data Innovators Gathering Challenge of 2015 [9], and state-of-the-art approaches in the literature [3] have shown that the intensities of pixels in the assembly-based grayscale images work well when used in conjunction with other features.

[7] `https://en.wikibooks.org/wiki/X86_Assembly/X86_Architecture#x86_Architecture`

[8] 6

[9] `https://www.kaggle.com/c/malware-classification`

https://t.me/learningnets

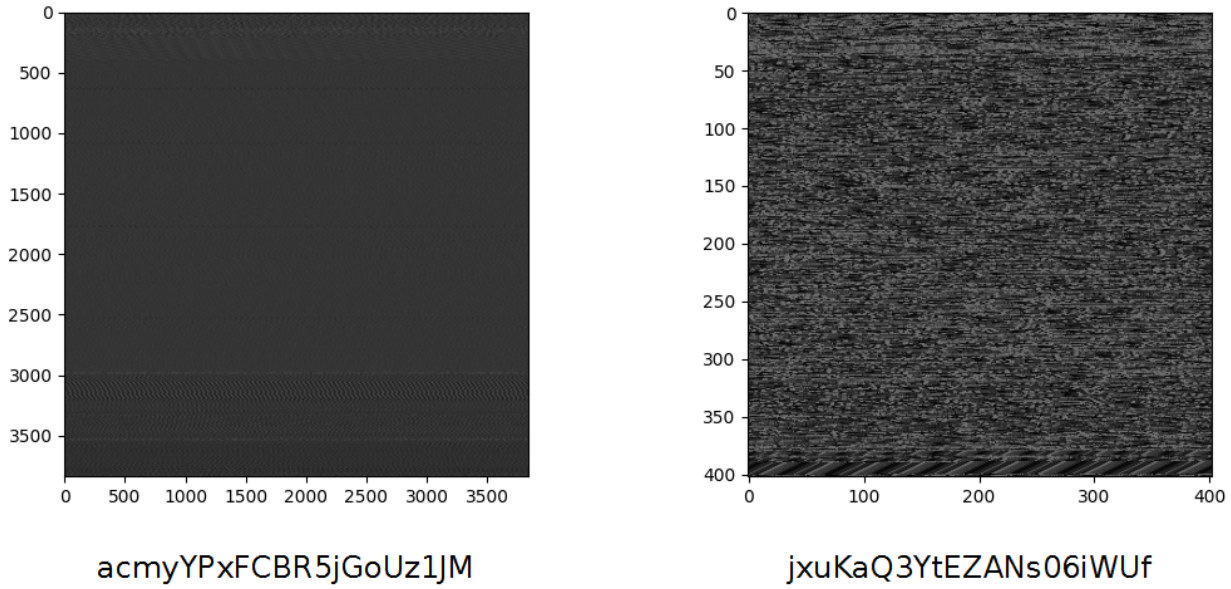


Figure 6: Grayscale image representation of malware's assembly language source code.

Section features (*ASM_SEC*). A Portable Executable (PE) file consists of various predefined sections, e.g. .text, .data, .bss, .rdata, .edata, .idata, .rsrc, .tls, and .reloc. However, malware authors usually employ obfuscation techniques such as packing, which might modify the default sections and create new sections. For example, the UPX packer (Ultimate Packer for Executables), one of the most popular and well-known packer, typically creates two sections named UPX0 and UPX1. For this reason, we extracted different characteristics from the sections in the PE executables related to the proportion of code in those sections with respect to the whole file. The full list of section-based features is presented in Table 3.

Miscellaneous features (*ASM_MISC*). This feature category consists of the frequency of 95 manually chosen keywords from the assembly language source code [2], mostly consisting of strings and dlls. The full list of miscellaneous features is listed in the project's repository [10].

4.2 Deep Learning Architectures

Our system also automatically extracts features from raw data through deep learning. More specifically, it relies on the extraction of N-gram like features from both the hexadecimal representation of malware's binary content (byte N-grams) and its assembly language source code (opcode N-grams), shapelet-based features from malware's structural entropy, and texture patterns from malware's binary content represented as grayscale images.

4.2.1 Texture-based Features from the Grayscale Image Representation of Malware

Instead of using well-know feature extractors, e.g. Haralick [29], Local Binary Patterns [30], etc, we implemented a convolutional neural network to automatically learn discriminative features (*BYTE_IMG_CNN*) from the grayscale image representation of malware [28] given its superior performance for the task of malware classification [11]. In their work, Gibert et al. [11] compared the performance of CNN classifiers against various state-of-the-art feature extractors, including Haralick, Local Binary Pattern, PCA, and GIST, for the task of malware classification and the results show a 2.31%, 1.88%, 1.99%, and a 1.34% increase with respect to the classification performance of the classifiers trained with the LBP, Haralick, PCA and GIST features, respectively.

The architecture of our choice is the one used in the work of Gibert et al. [11]. In particular, this architecture consists of three convolutional blocks composed by a convolutional layer, max-pooling layer and a normalization layer, followed by two fully-connected layers and a softmax layer. See Figure 7. Notice that the resulting grayscale images have been resized to size equals 255×255 using the Lanzcos filter as in their work. For more details about the architecture we refer the reader to the original article [11].

[10] 6

https://t.me/learningnets

Table 3: List of features in the *ASM_SEC* category [2].

Feature Name	Description
.bss	Total number of lines in the .bss section
.data	Total number of lines in the .data section
.edata	Total number of lines in the .edata section
.idata	Total number of lines in the .idata section
.rdata	Total number of lines in the .rdata section
.rsrc	Total number of lines in the .rsrc section
.text	Total number of lines in the .text section
.tls	Total number of lines in the .tls section
.reloc	Total number of lines in the .reloc section
.bss_por	Proportion of .bss section to the whole file
.data_por	Proportion of the .data section to the whole file
.edata_por	Proportion of the .edata section to the whole file
.idata_por	Proportion of the .idata section to the whole file
.rdata_por	Proportion of the .rdata section to the whole file
.rsrc_por	Proportion of the .rsrc section to the whole file
.text_por	Proportion of the .text section to the whole file
.tls_por	Proportion of the .tls section to the whole file
.reloc_por	Proportion of the .reloc section to the whole file
Num_Section	Total number of sections
Unknown_Sections	Total number of unknown sections
Known_Sections_lines	Total number of lines in known sections
Unknown_Sections_lines	Total number of lines in unknown sections
Known_Sections_por	Proportion of known sections to all sections
Unknown_Sections_por	Proportion of the unknown sections to all sections

4.2.2 Shapelet-based Features from Malware's Structural Entropy Representation

To complement the statistical hand-crafted features (*BYTE_ENT*) defined in Section 4.1.1 we implemented a convolutional neural network architecture to learn features from the structural entropy representation of malware's binary content [10]. Cf. Figure 8. As it has been documented in Gibert et al. [10] and Baysa et al. [33], the structural entropy of malware belonging to the same family is very similar while distinct from the structural entropy representation of malware belonging to the other families. The rationale behind applying convolutional layers to the stream of entropy values is to learn hierarchical shapelet-like features from this kind of representation. A shapelet is a subsequence of a time series which is representative of a class (family). The idea behind is to distinguish the samples of malware belonging to different families by their local variations instead of their global structure However, brute-force shapelet-based approaches are computationally expensive [34] and thus, we used convolutional layers to learn which are the optimal shapelets without exploring all possible candidates. More specifically, the convolutional layers can be seen as detection filters for specific subsequences present in the structural entropy of malware, going from low-level features in the first layers to increasingly complex features in the last layers.

4.2.3 Byte and Opcode N-Gram like Features

One of the most common types of features to classify malware are N-grams [35]. An N-gram is a contiguous sequence of n items from a given sequence of text. The rationale behind using N-grams to detect malware is that malicious software have structure, and N-grams work by capturing this structure (certain combinations of N-grams are more likely in samples belonging to some malware families than others. For the task of malware detection and classification, N-grams can be extracted from the hexadecimal representation of malware's binary content and from the assembly language source code, also known as byte N-grams and opcode N-grams, respectively.

Byte N-grams and opcode N-grams refer to the unique combination of every n consecutive bytes and opcodes as individual features, respectively. As a result, N-gram based approaches construct a vector of features, where each feature in the vector indicates the number of appearances of a particular N-gram. Consequently, the length of the feature vector depends on the number of unique N-grams, which increases with N. This leads to two main shortcomings that limit the applicability of N-grams in a real-world scenario (when $N > 3$). First, the resulting feature vector is very large as the model has to store the count for all N-grams occurring in the dataset. Second, the feature vector is very sparse, most features have zero values, increasing the space and time complexity of the resulting models. Generally, if there are

https://t.me/learningnets

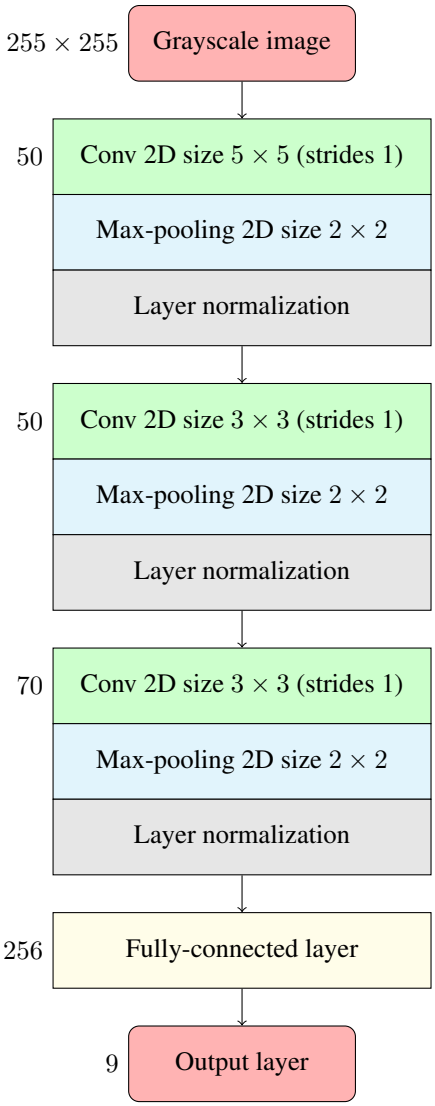


Figure 7: Convolutional neural network architecture for classifying malware's binary content represented as a grayscale image.

too many features,the machine learning models tend to fit the noise in the training data. This is known as overfitting, and it results in poor generalization on newer data. As a result, machine learning approaches based on N-grams for the task of malware detection and classification have limited the size of N to size 3 or 4, and applied feature selection and dimensionality reduction techniques to reduce the dimensionality of the resulting feature vector.

Alternatively, Gibert et al. [8, 9] proposed a shallow convolutional neural network architecture to extract N-gram like features from malware's assembly language source code and binary content. Cf. Figure 9. This is achieved by convolving various filters of different sizes k, where k $\in \{3, 5, 7\}$ in our case, which indicates the number of opcodes and bytes to which is applied. Afterwards, a global max-pooling layer is applied to retrieve the maximum activation for each feature map independently of their position in the input sequence. This can be seen as if a particular N-gram has been found in the input sequence. Alternatively, one could employ a global average-pooling layer to retrieve the average of the activations for each feature map. However, in our experiments the global max-pooling layer achieved higher classification results. For more details about the architecture we refer the reader to the original publications [8, 9].

https://t.me/learningnets

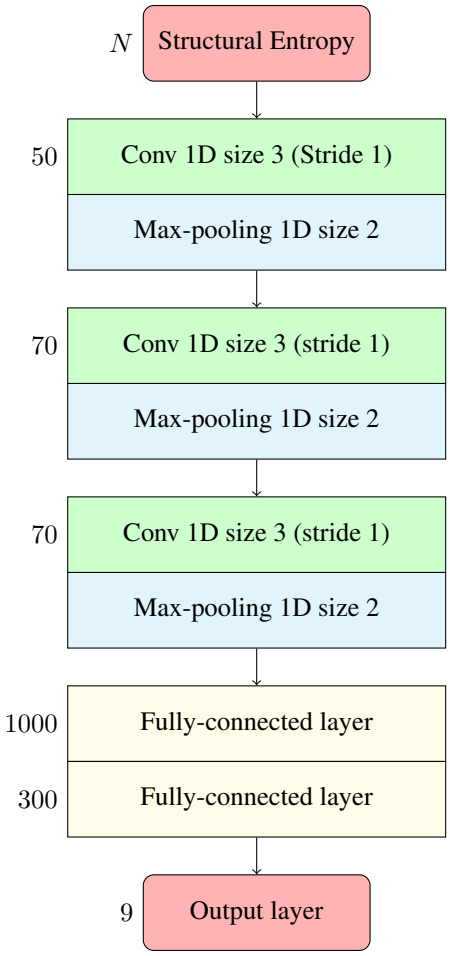


Figure 8: Convolutional neural network architecture for classifying malware's binary content represented as a stream of entropy values.

4.3 Feature Fusion

At this point, the executable is represented as various feature vectors $\overrightarrow{v}$, $\overrightarrow{w}$, $\overrightarrow{x}$, ..., $\overrightarrow{z}$, one for each type of features, e.g. BYTE_MD, BYTE_1G, BYTE_ENT, etc, that provide an abstract view of their content. To integrate the feature vectors, our system employs an early fusion mechanism to create a joint representation of the features from multiple modalities. This fusion mechanism combines the various feature vectors by concatenating them into a single feature vector. Cf. Figure 10. Afterwards, a single model is trained to learn the correlation and interactions between the features of each modality. Given various feature vectors $\overrightarrow{v}$, $\overrightarrow{w}$, $\overrightarrow{x}$, ..., $\overrightarrow{z}$ containing the different types of features, the prediction of the final model, denoted as h, can be written as:

$$p = h\left([\overrightarrow{v}, \overrightarrow{w}, ..., \overrightarrow{z}]\right) = \left([v_1, v_2, ..., v_i, w_1, w_2, ..., w_j, ..., z_1, z_2, ..., z_k]\right)$$

4.4 Gradient Boosting Trees

For classification purposes, we trained our machine learning model using XGBoost [36], a parallel implementation of the gradient boosting tree classifier. The impact of XGBoost has been widely recognized in many machine learning and data mining competitions, where approximately half of the winning solutions used XGBoost. Next, we review gradient boosting tree algorithms. For a more detailed description of gradient boosting trees and XGBoost we refer the reader to the original article [36].

Boosting is an ensemble learning technique for building a strong classifier in the form of an ensemble of weak classifiers. When the weak learners are decision trees, the resulting algorithm is called boosted trees. Gradient Boosting Trees (GBT) is an ensemble learning technique that builds one tree at a time, where each new tree attempts to minimize the

https://t.me/learningnets

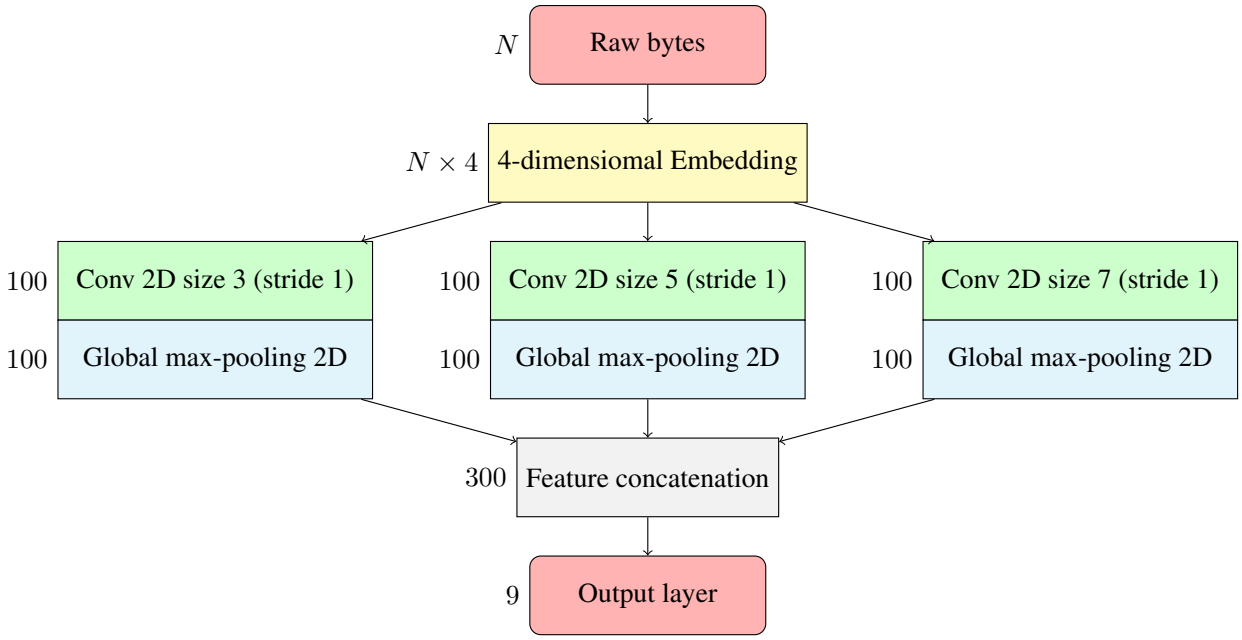


Figure 9: Shallow convolutional neural network architecture to extract N-gram like features from the hexadecimal representation of malware's binary content. Notice that the only difference with respect to the architecture that extracts N-gram like features from the assembly language source code of malware is the input. Instead of the raw bytes sequence, the input is the sequence of opcodes extracted from the assembly code.

errors of the previous tree. This is achieved by combining sequentially the weak learners in a way that each new learner fits to the residuals from the previous steps so that the model improves. Afterwards, the final model aggregates the results from each step to build the strong learner. Notice that to detect the residuals a loss function should be used, e.g. the logarithmic loss for classification tasks. Subsequently, adding many trees sequentially, and each one focusing in the errors from the previous one, makes boosting a highly efficient and accurate model for classification tasks.

5 Evaluation

5.1 The Microsoft Malware Classification Challenge Dataset

The task of Windows malware detection has not received the same attention by the research community as other domains, where rich datasets exist [15]. In addition, the copyright laws that prevent sharing benign software has exacerbated this situation. As a result, no dataset containing benign and malicious software is available to the public for research. The only benchmark available for malware detection is the Ember dataset [37], which provides a collection of features from PE files. However, the raw binaries are not included and thus, the application of deep learning or the extraction of new features from the executables is not possible. To make things worse, even as malicious binaries may be obtained for internal use through web services such as VirusTotal, its subsequent sharing of the binary or the vendor antimalware labels assigned are prohibited. Furthermore, unlike other domains where data samples may be labeled quickly and, in most cases by a non-expert, determining if a file is malicious or not is a very time consuming and complex process, even for security experts. This issues makes it impossible to meaningfully compare accuracy across works because different datasets with distinct labeling procedures are used from one work to another. For this reason, instead of creating our own private dataset, we decided to evaluate our system with the data provided by Microsoft for the Big Data Innovators Gathering Challenge of 2015, the only high-quality public labeled benchmark available for malware classification research [17].

This dataset has become the standard benchmark to evaluate machine learning approaches for malware classification and currently, it is publicly available in the Kaggle platform [11]. It includes half a terabyte of data consisting of 10868 samples for training and 10873 samples for testing. To sum up, the dataset contains samples representing 9 different malware families, where each sample has associated two files: (1) the hexadecimal representation of malware's binary

[11] `https://www.kaggle.com/c/malware-classification`

https://t.me/learningnets

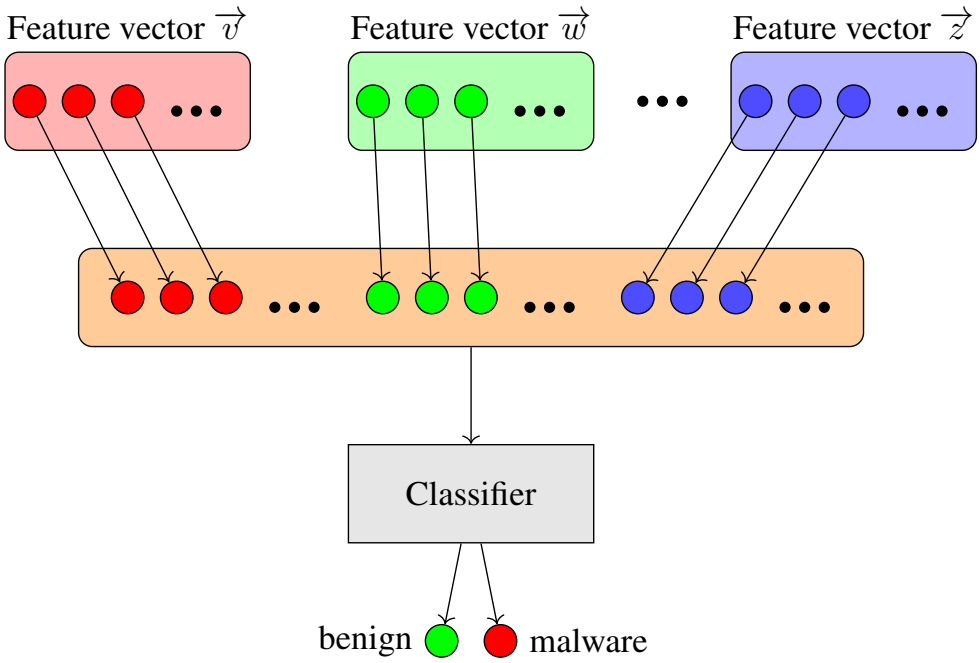


Figure 10: Early fusion strategy

content and (2) its corresponding assembly language source code, generated using the IDA Pro disassembler [12]. Cf. Figures 2 and 3. The families represented are the following: (1) Ramnit, (2) Lollipop, (3) Kelihos_ver3, (4) Vundo, (5) Simda, (6) Tracur, (7) Kelihos_ver1, (8) Obfuscator.acy and (9) Gatak. Following, a brief description of the behavior of each family is provided according to Microsoft Security Intelligence [13].

Ramnit. This worm family is reported to have the capability to steal your sensitive information such as saved FTP credentials and browser cookies, and can spread via removable drives.

Lollipop. This adware family is reported to display ads in your browser as you navigate the Internet. In addition, it can also redirect your search engine results, monitor your PC and download other applications.

Kelihos_ver3. This backdoor family is reported to install on the system in order to download other components, and includes a backdoor that gives the attacker further control over the affected system. Computer systems affected by this malware were used as bots in the Kelihos botnet, now deceased. Version 3 of the software.

Vundo. This trojan family is reported to cause popups and advertising for rogue anti-spyware programs.

Simda. This backdoor family is reported to install on the system to give an attacker remote control of the system. In addition, this malware is reported to steal personal and system data, take screenshots and download additional files.

Tracur. This family of malware is reported to redirect you Internet search queries to malicious URLs to download and install other malware.

Kelihos_ver1. This backdoor family is reported to install on the system in order to download other components, and includes a backdoor that gives the attacker further control over the affected system. Computer systems affected by this malware were used as bots in the Kelihos botnet, now deceased. Version 1 of the malicious software.

Obfuscator.ACY. This family of malware comprises software that has been obfuscated, that is, software that has tried to hide its behavior or purpose so that anti-malware engines do not detect it. The software that lies underneath this obfuscation can have any purpose.

Gatak. This trojan family is reported to silently download and install other software without the user's consent.

[12] `https://www.hex-rays.com/products/ida/`
[13] `https://www.microsoft.com/en-us/wdsi`

https://t.me/learningnets

As it can be observed in Table 4, the dataset is very imbalanced and the class distribution is not uniform among families, i.e. the number of samples belonging to some families significantly outnumber the number of samples belonging to the remaining families.

Table 4: Class distribution in the Microsoft dataset [17]

Family	#Samples	Type
Ramnit	1541	Worm
Lollipop	2478	Adware
Kelihos_ver3	2942	Backdoor
Vundo	475	Trojan
Simda	42	Backdoor
Tracur	751	TrojanDownloader
Kelihos_ver1	398	Backdoor
Obfuscator.ACY	1228	Obfuscated malware
Gatak	1013	Trojan

5.2 Experimental Setup

The system has been deployed on a machine with an Intel Core i7-7700k CPU, 2xGeforce GTX1080Ti GPUs and 64Gb RAM. To implement the convolutional neural network architectures it has been used Tensorflow [38].

The generalization performance of our multimodal approach has been estimated using k-fold cross validation, with k equals to 10. K-fold cross validation is a model validation technique to assess how accurately a predictive model will perform in practice. In k-fold validation, the dataset is partitioned into k folds of equal size. Then, the following procedure is followed for each one of the k folds:

- A model is trained using $k-1$ of the folds as training data.
- The resulting model is validated on the remaining fold of the data.

Afterwards, the performance measure reported is the average of the values achieved for each fold.

Next, are presented the performance metrics and the experiments carried out to evaluate our multimodal approach. In particular, the experiments have been designed to evaluate the classification performance of each individual subset of features, to compare the performance of the model with and without hand-crafted and deep features, and to evaluate the fusion of features using early fusion and the forward stepwise selection algorithm. Lastly, our approach has been compared with the state-of-the-art approaches in the literature.

5.2.1 Performance Metrics

Regarding the performance metrics used to evaluate our approach, we will report two metrics, the accuracy and the logarithmic loss.

The accuracy is simply the fraction of correct predictions. Formally, accuracy is defined as follows:

$$accuracy = \frac{Number\ of\ correct\ predictions}{Total\ number\ of\ predictions}$$

However, accuracy alone is not a good evaluation metric to assess the robustness of machine learning models in datasets where there exist a large class imbalance. Subsequently, the multi-class logarithmic loss (logloss) has been used to to assess the performance of the predictions. The logarithmic loss is the cross entropy between the distribution of true labels and the predicted probabilities. Formally, it is defined as follows:

$$logloss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{i,j}\ log(p_{i,j})$$

where N is the number of observations, M is the number of class labels, log is the natural logarithm, $y_{i,j}$ is 1 if the observation i is in class j and 0 otherwise, and $p_{i,j}$ is the predicted probability that observation i is in class j. Notice that for the test set it is only provided the multi-class logarithmic loss or logloss. This is because the labels of the samples in the test set are not provided and to assess the performance of a given model you need to submit the predicted probabilities to Kaggle.

https://t.me/learningnets

5.2.2 Individual Subsets of Features Performance Analysis

Table 5 presents the 10-fold cross validation accuracy and logloss achieved by the machine learning models trained using only an individual subset of features in the training and test sets. XGboost has various hyperparameters that are completely tunable. Cf. Table 6. For now, we will skip the details of the hyperparameters and set all of them to their baseline value. Table 5 gives us some clues about the discriminative power of each feature category. There are three feature categories that have very low accuracy and high logarithmic loss in comparison to the remaining ones. The feature categories are the following: *BYTE_MD*, *ASM_MD*, and *ASM_PIXEL*. This means that this type of features alone are not enough to correctly categorize malware into families with high accuracy, but as described in Section 5.2.3, some of them are very valuable in the construction of the multimodal boosted decision trees. Thus, we decided to keep this features in our final model. However, the classification performance of the models trained on a single feature category cannot compete with the state-of-the-art approaches in the literature. Thus, in the following sections various mechanisms to fuse the different feature categories are studied. See Section 5.2.6.

Table 5: Classification performance of each feature category

	Train		Test
Feature Category	Accuracy	Logloss	Logloss
BYTE_MD	0.8634	0.4066	0.3975
BYTE_1G	0.9835	0.0602	0.0386
BYTE_ENT	0.9778	0.0788	0.0728
BYTE_HARALICK	0.9741	0.0892	0.0833
BYTE_LBP	0.9791	0.0782	0.0608
BYTE_IMG_CNN	0.9464	0.0139	0.1066
BYTE_ENT_CNN	0.9703	0.1118	0.1291
BYTE_NGRAMS_CNN	0.9756	0.0074	0.0302
ASM_MD	0.9072	0.3323	0.2807
ASM_OPC	0.9598	0.0090	0.0263
ASM_PIXEL	0.7661	0.5346	0.5442
ASM_REG	0.9334	0.0168	0.0533
ASM_SYM	0.8905	0.0288	0.0942
ASM_API	0.9840	0.0591	0.0546
ASM_DD	0.9851	0.0606	0.0411
ASM_SEC	0.9879	0.0749	0.0329
ASM_MISC	0.9926	0.0282	0.0206
ASM_NGRAMS_CNN	0.9917	0.0299	0.0356

One benefit of gradient boosting trees is that in order to construct the decision trees, they have to explicitly calculate the importance for each feature in the dataset. Feature importance is a score that indicates how useful or valuable each feature has been in the construction of the boosted decision trees. The more a particular feature has been selected to make key decisions in the decision trees, the higher its relative importance. For a single decision tree, importance is calculated by the amount each feature split point improves the performance measure, weighted by the number of observations the node is responsible for. In the case of XGBoost, this performance measure is the purity (Gini index) used to select the split points. These feature importances are then averaged across all of the decision trees within the model to retrieve their final importance scores. Accordingly, we included the top 20 most important features for each feature category in A.

Table 6: List of hyperparameters and their values

Hyperparameter	Baseline Value	Best Value	Description
eta	0.2	0.1	Step size shrinkage.
max_depth	5	3	Maximum depth of a tree.
gamma	0.0	0.0	Minimum loss reduction required to make a further partition on a leaf node of the tree.
min_child_weight	1	1	Minimum sum of instance weight (hessian) needed in a child.
colsample_bytree	1.0	1.0	It is the subsample ratio of columns when constructing each tree.
subsample	1.0	1.0	Subsample ratio of the training instances.

https://t.me/learningnets

5.2.3 Early-fusion Performance

This second experiment aims at studying the performance of the boosted trees models when trained with various feature categories combined using early-fusion with and without the deep features. Subsequently, we trained the following models:

Hex-based hand-crafted features. It refers to the model trained using the features manually extracted from the hexadecimal view of malware. Feature categories: *BYTE_MD*, *BYTE_1G*, *BYTE_ENT*, *BYTE_HARALICK*, *BYTE_LBP*.

Hex-based hand-crafted & deep features. It refers to the model trained using both the hand-crafted and deep features extracted from the hexadecimal view of malware. Feature categories: *BYTE_MD*, *BYTE_1G*, *BYTE_ENT*, *BYTE_LBP*, *BYTE_HARALICK*, *BYTE_IMG_CNN*, *BYTE_ENT_CNN*, *BYTE_NGRAMS_CNN*.

Assembly-based hand-crafted features. It refers to the model trained using the features manually extracted from the assembly view of malware. Feature categories: *ASM_MD*, *ASM_OPC*, *ASM_PIXEL*, *ASM_REG*, *ASM_SYM*, *ASM_API*, *ASM_DD*, *ASM_SEC*, *ASM_MISC*.

Assemly-based hand-crafted & deep features. It refers to the model trained using both the hand-crafted and deep features extracted from the assembly view of malware. Feature categories: *ASM_MD*, *ASM_OPC*, *ASM_PIXEL*, *ASM_REG*, *ASM_SYM*, *ASM_API*, *ASM_DD*, *ASM_SEC*, *ASM_MISC*, *ASM_NGRAMS_CNN*.

Hand-crafted features. It refers to the model trained using the features manually extracted from both the hexadecimal and assembly view of malware. Feature categories:
BYTE_MD, *BYTE_1G*, *BYTE_ENT*, *BYTE_HARALICK*,
BYTE_LBP, *ASM_MD*, *ASM_OPC*, *ASM_PIXEL*, *ASM_REG*, *ASM_SYM*, *ASM_API*, *ASM_DD*, *ASM_SEC*, *ASM_MISC*.

Deep features. It refers to the model trained using the features transferred from the deep learning models. Feature categories: *BYTE_IMG_CNN*, *BYTE_ENT_CNN*,
BYTE_NGRAMS_CNN, *ASM_NGRAMS_CNN*.

Hand-crafted & deep features. It refers to the model trained using both the hand-crafted and deep features extracted from the hexadecimal and the assembly view of malware. Feature categories: *BYTE_MD*, *BYTE_1G*, *BYTE_ENT*, *BYTE_HARALICK*, *BYTE_LBP*, *ASM_MD*, *ASM_OPC*,
ASM_PIXEL, *ASM_REG*, *ASM_SYM*, *ASM_API*, *ASM_DD*, *ASM_SEC*, *ASM_MISC*, *BYTE_IMG_CNN*, *BYTE_ENT_CNN*, *BYTE_NGRAMS_CNN*, *ASM_NGRAMS_CNN*.

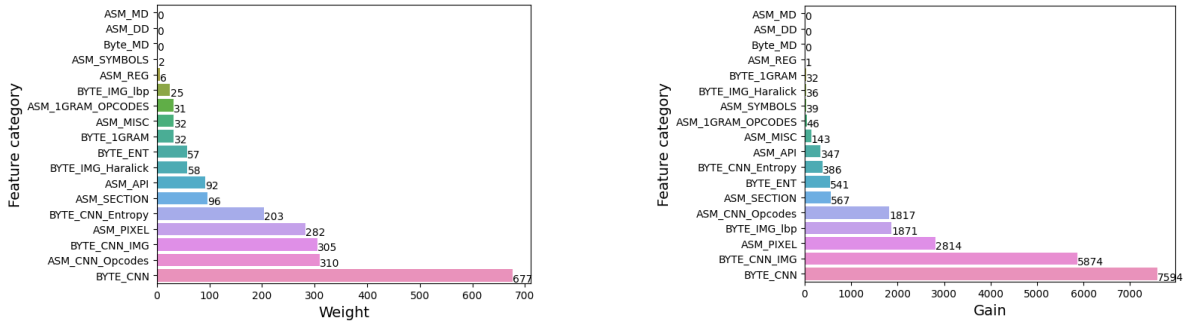
Table 7: Classification performance of the early-fusion models.

	Train		Test
Feature Category	Accuracy	Logloss	Logloss
Hex-based hand-crafted features	0.9962	0.0138	0.0207
Hex-based hand-crafted & deep features	0.9958	0.0154	0.0193
Assembly-based hand-crafted features	0.9978	0.0100	0.0063
Assembly-based hand-crafted & deep features	0.9978	0.0069	0.0052
Hand-crafted features	0.9976	0.0088	0.0073
Deep features	0.9954	0.0154	0.0194
Hand-crafted & deep features	0.9987	0.0059	0.0086

Table 7 presents the classification performance of the aforementioned methods. It can be observed that the models trained with both hand-crafted and deep features achieve higher accuracy and lower logarithmic loss than their hand-crafted counterpart. However, the more complex model, the lower its performance. It can be observed that the model trained with hand-crafted and deep features from both the hexadecimal and the assembly view of malware, contains 956 more features than the hand-crafted model, making it more prone to overfitting. Overfitting occurs when the classification model models too well the training data, that is, the details and noise in the training data, to the extent that it negatively impacts the performance of the model on new data (fails to generalize to unseen data). For this reason, the hyperparameters of the XGBoost models were tuned to avoid overfitting. In particular, we performed an heuristic search over the hyperparameters listed in Table 6. Following, is provided a brief description of each hyperparameter:

- eta. Eta is the learning rate of our gradient boosted trees model. It indicates how much we update the prediction with each successive tree. The lower the eta is, the more conservative the boosting process will be.
- max_depth. It refers to the maximum depth of a tree.The large max_depth is, the more complex and likely to overfit the model will be.

https://t.me/learningnets



(a) Number of times a feature category has been used to split the data across all trees.

(b) Average gain across all splits the feature categories have been used in.

Figure 11: Importance of each feature category.

- gamma. The gamma is the minimum loss reduction required to make a further partition on a leaf node of the tree. A larger gamma makes the algorithm more conservative.
- min_child_weight. It refers to the minimum sum of instance weight needed in a leaf. If the weight for every instance is 1, it directly indicates to the minimum number of instances needed in a node.
- colsample_bytree. It indicates the subsample ratio of features when constructing each tree.
- subsample. It refers to the subsample ratio of the training instances. For instance, setting its value to 0.5 means that XGBoost would randomly sample half of the training data prior to growing trees.

By training the boosted trees model using the best hyperparameters defined in Table 6 we have been able to fine-tune the algorithm and reduce the logarithmic loss achieved on the test set from 0.008860 to 0.00636, the second lowest loss reported in any published article by a single model so far, only behind the model trained using both the hand-crafted and deep features extracted from the assembly view of malware, which achieved a logloss of 0.00516. Notice that state-of-the-art approaches in the literature [2, 3] employ one or more ensemble learning techniques such as bagging. However, the fact that the model trained with only the assembly features achieves better results than the model using all features indicates that there are at least one or more subsets of features that do not apply to new data and negatively affect the model's ability to generalize. In addition, as it can be observed in Figure 11, the number of times a particular feature occurs in the trees of the model greatly varies between categories. For instance, the model trained with all features has used features from the *BYTE_CNN*, *BYTE_CNN_IMG*, *ASM_CNN_OPCODES* and *ASM_PIXEL* categories 840, 291, 310 and 300 times, respectively. On the other hand, metadata features (*ASM_MD* and *BYTE_MD*) and data define features (*ASM_DD*) have not been used for building the final model. However, it is not possible to know if it is because of their unimportance or because the combination of other features achieves the same effect. Given this circumstances, we decided to use various feature selection techniques to select the most discriminant subset of features.

5.2.4 Feature Selection

Fusing all subsets of features into a single feature vector produces a feature vector of size equals to 3349, quite large compared to the number of training instances, which is 10868. However, as it can be observed in Section 5.2.3, a feature vector with all features is not optimal and produces an overfitted model, i.e. a model with poor generalization performance. What is happening is known as the Hughes phenomenon. That is, as the number of features grows, the classifier's performance increases until it reaches the optimal number of features. Afterwards, the performance of the classifier decreases as the dimensionality of the feature space increases, under the assumption that the number of training samples remains fixed.

In theory, the classification algorithm used to train our models (see Section 4.4) naturally selects which features are most important when constructing the decision trees. However, as it can be observed in Section 5.2.3, when the model is trained using all features it ends up overfitting the training data and performing poorly on the test set. In consequence, univariate feature selection has been investigated to select a subset of the features that have a major statistically significant relationship with the target variable (the malware families) based on various univariate statistical tests. More specifically, the best subset of features has been selected using the Chi-squared [39], the ANOVA F-value [40], and the Mutual Information [41] score functions. Unfortunately, as it can be observed in Table 8, the models trained using univariate feature selection performed poorly in comparison to the models presented in Section 5.2.3. Our intuition is that, as univariate feature selection works by selecting the features that have a significant statistical relationship with the

https://t.me/learningnets

target variable (class or family) and does not take into account the relationship between features, it might be discarding features that alone are not very discriminant but work well in combination with other features. In consequence, a wrapper method to select the best subset of features has been proposed.

Table 8: List of models trained using the K best features according to various univariate metrics and their evaluation with XGBoost.

	Chi square			ANOVA f-value			Mutual Information		
	Train		Test	Train		Test	Train		Test
K	Accuracy	Logloss	Logloss	Accuracy	Logloss	Logloss	Accuracy	Logloss	Logloss
20	0.4987	1.3673	1.3654	0.6138	0.9078	0.9094	0.9961	0.0163	0.0124
50	0.5627	1.2434	1.2304	0.6138	0.9086	0.9092	0.9969	0.0135	0.0100
100	0.9544	0.1323	0.1355	0.6137	0.9087	0.9089	0.9967	0.0126	0.0089
200	0.9977	0.0095	0.0097	0.7187	0.6430	0.6430	0.9970	0.0116	0.0092
500	0.9980	0.0071	0.0097	0.9956	0.0144	0.0196	0.9975	0.0108	0.0096
1000	0.9985	0.0065	0.0117	0.9983	0.0068	0.0096	0.9983	0.0073	0.0079
1500	0.9984	0.0061	0.0094	0.9985	0.0061	0.0089	0.9983	0.0059	0.0088
2000	0.9988	0.0062	0.0093	0.9986	0.0057	0.0088	0.9983	0.0056	0.0084
2500	0.9988	0.0057	0.0085	0.9988	0.0058	0.0097	0.9987	0.0058	0.0073

5.2.5 Forward Stepwise Selection Technique

Starting with a model containing no features, the forward stepwise selection algorithm gradually increases the feature set by adding subsets of features, one by one. To determine the subset of features to add at each step, it is used the logarithmic loss achieved on the validation set using K-fold cross validation, where K equals 3. That is, at each step, we add to the feature set the subset of features that produce the minimum value of the logarithmic loss. This process stops when adding more subsets of features does not decrease the value of the logarithmic loss. The rationale behind using a smaller K than in Section 5.2.3 is that it greatly reduces the training computational time as it reduces to three the number of models to be trained. Unfortunately, as it can be observed in Table 9, although adding more subsets of features decrease the logarithmic loss on the validation set, when the model is evaluated on the test set, better results have been achieved by a model with a smaller subset of features. In this case, the lowest logarithmic loss on the test set has been achieved by a model trained using only two feature subsets: (1) *ASM_NGRAMS_CNN*, and (2) *ASM_PIXEL*.

Table 9: Gradual addition of feature categories using the forward stepwise selection algorithm.

		Train		Test
Feature Category	Number of features	Accuracy	Logloss	Logloss
C1: ASM_NGRAMS_CNN	300	0.9928	0.0270	0.0356
C2: C1 + ASM_PIXEL	1100	0.9979	0.0079	0.0072
C3: C2 + BYTE_IMG_CNN	1356	0.9983	0.0071	0.0110
C4: C3 + ASM_API	2150	0.9986	0.0065	0.0094
C5: C4 + BYTE_1G	2405	0.9989	0.0061	0.0101
C6: C5 + BYTE_NGRAMS_CNN	2705	0.9989	0.0060	0.0092
C7: C6 + ASM_OPC	2798	0.9987	0.0059	0.0092
C8: C7 + BYTE_LBP	2906	0.9987	0.0058	0.0091
C9: C8 + ASM_MD	2908	0.9987	0.0057	0.0076
C10: C9 + ASM_REG	2946	0.9988	0.0057	0.0081

5.2.6 Comparison with the State-Of-The-Art

Given that previous feature selection techniques have not achieved better results than the model trained using only the assembly features, we built the final model using as features those extracted from the assembly language source code, *ASM_MD*, *ASM_OPC*, *ASM_PIXEL*, *ASM_REG*, *ASM_SYM*, *ASM_API*,
ASM_DD, *ASM_SEC*, *ASM_MISC*, *ASM_NGRAMS_CNN*, plus three feature subsets from the binary content, *BYTE_MD*,
BYTE_1G, and *BYTE_NGRAMS_CNN*. The reason is that adding any other of the feature subsets deteriorates the classification performance of the system.

https://t.me/learningnets

Next, our approach is compared with the state-of-the-art methods in the literature that are based on feature engineering and deep learning. Only approaches that have evaluated their methods using the Microsoft benchmark and employ K-fold cross validation have been selected for comparison. This has been done to ensure fairness when comparing the methods. Take into account that methods evaluated on different datasets are not directly comparable, and most often than not, the source code is not available online or it does not work properly, i.e. the parameters have been optimized for their own dataset, there are missing libraries, etc. For this reason, the results presented in Table 10 are those published in their original publications, without modifications.

Table 10: Comparison with the state-of-the-art methods on the Microsoft Malware Classification Challenge benchmark. Those approaches that their authors have not tested their performance on the test set or did not make public the K-fold cross validation accuracy or logarithmic loss appear with a '-' mark. Approaches with a '*' mark indicate that they performed 5-fold cross validation instead of 10-fold cross validation. Approaches that have not performed K-fold cross validation but have used a single hold-out validation set are marked with "**".

		10-fold Cross Validation		Test
Method	Input	Accuracy	Logloss	Logloss
Narayanan et al. [4]	Grayscale images	0.9660	-	-
Kebede et al. [42]**	Grayscale images	0.9915	-	-
Gibert et al. [11]	Grayscale images	0.9750	-	0.1845
Kalash et al. [43]**	Grayscale images	0.9852	-	0.0571
Liu et al. [44]	Grayscale images	0.9900	-	-
Vinayakumar et al. [45]	Grayscale images	0.9630	-	-
Lo et al. [46]	Grayscale images	-	-	0.0361
Qiao et al. [47]	Grayscale images	0.9876	-	-
Sudhakar and Kumar[24]	Grayscale images	0.9853	-	-
Xiao et al. [23]	Grayscale images	0.9894	-	-
Çayır et al. [48]**	Grayscale images	0.9926	-	-
Lin and Yeh[49]	Grayscale images	0.9632	-	-
Yuan et al. [50]	Markov images	0.9926	0.0518	-
Kim et al. [51]	RGB images	0.9574	-	-
Jiang et al. [21]	RGB images	0.9973	-	0.0220
Zhang et al. [52]	RGB images	0.9934	-	-
Gibert et al. [10]	Structural entropy	0.9828	-	0.1244
Xiao et al. [53]	Structural entropy	0.9972	-	0.0314
Yan et al. [54]	Control flow graph	0.9925	0.0543	-
Hu et al. [55]	Opcode 4-grams	0.9930	-	0.0546
Gibert et al. [8]	Opcode sequence	0.9917	-	0.0244
Gibert et al. [56]	Opcode sequence	0.9913	-	0.0419
McLaughlin et al. [57]	Opcode sequence	0.9903	-	0.0515
Yousefi-Azar et al. [14]	Byte sequence	0.9309	-	-
Drew et al. [58]	Byte sequence	0.9741	-	0.0479
Raff et al. [19]	Byte sequence	0.9641	-	0.3071
Krčál et al. [20]	Byte sequence	0.9756	-	0.1602
Kim and Cho[59]	Byte sequence	0.9747	-	-
Le et al. [60]	Compressed byte sequence	0.9820	-	0.0774
Gibert et al. [13]	Compressed byte sequence	0.9861	-	0.1063
Messay-Kebede et al. [61]	Opcode statistics and byte sequence	0.9907	-	-
Gibert et al. [62]	Opcode and byte sequences	0.9924	-	-
Gibert et al. [63]	API calls, Opcode and byte sequences	0.9975	-	-
Gao et al. [22]*	Hand-crafted features	0.9969	-	-
Ahmadi et al. [2]*	Hand-crafted features	0.9977	0.0096	0.0063
Zhang et al. [3]	Hand-crafted features	0.9976	-	0.0042
Proposed system	**Hand-crafted and deep features**	**0.9981**	**0.0070**	**0.0040**

As it can be observed, our multimodal approach achieves the highest accuracy and lowest logarithmic loss in the training set using K-fold cross validation, and the lowest logarithmic loss in the test set, outperforming any machine learning approach presented in the literature so far. This has been possible thanks to the N-gram like features extracted by the shallow convolutional neural networks from both the hexadecimal representation of malware's binary content and its assembly language source code. This features are not only discriminant but also computationally inexpensive in

https://t.me/learningnets

comparison to traditional N-gram features, which require to manually enumerate all N-grams during training, and later reduce and select a smaller subset of characterizing N-grams by employing feature selection or dimensionality reduction techniques. In addition, our approach outperformed state-of-the-art approaches without employing sampling techniques, stacking and ensemble learning techniques as in Zhang et al. [2] and Ahmadi et al. [3].

6 Conclusions

In this paper, we present a novel multimodal approach that combines feature engineering and deep learning to extract features from both the hexadecimal representation of malware's binary content and its assembly language source code to achieve state-of-the-art performance in the task of malware classification. To the best of our knowledge, this research is the first application to combine feature engineering and deep learning using a simple, but yet effective, early fusion mechanism for the problem of malware classification. The success of our multimodal approach would not had been possible without the extraction of the N-gram like features using the shallow convolutional neural networks trained on the bytes and opcodes sequence representing malware's binary content and its assembly language source code, respectively, which provide a computational inexpensive way to extract long N-gram like features without having to exhaustively enumerate all N-grams during training. Furthermore, by fusing hand-crafted features and deep features we have been able to combine the strengths of both approaches, descriptive domain-specific features and the ability of deep learning to automatically learn to extract features from sequential data without relying on the experts' knowledge of the domain.

Reported results allow to assess its effectiveness with respect to the state-of-the-art for the task of malware classification, achieving the highest accuracy and lowest logarithmic loss reported in the Microsoft benchmark so far, while only using a single model to generate the final predictions.

6.1 Future Work

One future line of research could be the design and development of new architectures to extract features from malware's representation as grayscale images or its structural entropy as they perform below average in comparison with other features. In addition, a second line of research could be the analysis of the performance of various texture pattern extractors, to complement the Haralick and Local Binary Pattern feature extractors. Lastly, a third line of research could be the study of ensemble learning techniques such as blending and bagging, to build a stronger classifier by combining multiple models. In addition, the following limitations and open research questions should be dealt with in order to address the threat of malware.

6.2 Limitations of Deep Learning

With the vertical (numbers and volumes) and horizontal (types and functionality) expansion of malware threats during the last decade, malware detection has remained a hot topic as the number of yearly publications addressing the task demonstrate and it is still far from being solved. Recently, deep learning has started being adopted because of its ability to extract features from raw data [19, 20]. However, its application to the task of malware detection and classification has been limited so far due to the computational resources required to train such models. Take into account that depending on the type of input we might end up dealing with sequences of millions of time steps, which far exceeds the length of the input of any previous deep learning sequence classifier, leading to a huge consumption of the GPU memory in the first convolutional layers and the use of large filter widths and strides to balance the memory usage and the computational workload. This has forced researchers to explore alternatives to reduce and compress the information in the bytes sequence to make it a manageable problem [33, 10, 28, 60].

In addition, contrarily to computer vision, where the features extracted by deep learning aim to replace previous feature extractors, for the problem of malware detection the features learned through deep learning greatly differ from those usually extracted by domain experts to provide an abstract representation of the executable. More specifically, deep learning is only capable to substitute N-gram features, one of the many types of features usually extracted to characterize the executables. In consequence, machine learning systems trained on hand-engineered features tend to outperform deep learning approaches. For an extended description of common features extracted by domain experts we refer the readers to the work of Anderson et al. [37] and the references therein.

6.3 Open Research Questions

Persistent research efforts have been made using machine learning to address the threat of malware. However, machine learning models have proved to be susceptible to adversarial examples and concept drift.

https://t.me/learningnets

On the one hand, adversarial examples are inputs to a machine learning model that have been specifically crafted to exploit the weakness of the ML model in order to incorrectly label the malicious sample as benign. For instance, Hu et al. [64] proposed to craft the adversarial examples using a generative adversarial network (GAN) architecture named MalGAN, which learns which API imports should be added to the original sample to bypass detection. Sucio et al. [65] proposed to modify the existing bytes of a binary by appending content of benign executables and by using the Fast Gradient Method [66] to modify the random bytes appended at the end of the executables until the adversarial example evades detection. Demetrio et al. [67] proposed a functionality-preserving black-box optimization attack based on the injection of content of benign executables. To do so, they either inject code at the end of the file or within some newly-created section. Demetrio et al. [68] proposed three functionality-preserving attacks against ML-based models. These attacks, named Full DOS, Extend, and Shift, inject a crafted payload by manipulating the DOS header, extending it, and shifting the content in the first section, respectively. Given the success of the aforementioned attacks, countermeasures must be developed in other to mitigate the threat of malware. One mitigation strategy may be to consider features that are not affected by the changes at the section or byte level. Another countermeasure may be adversarial retraining. That is, including adversarial examples generated by the aforementioned attacks on the training set.

On the other hand, concept drift refers to the problem of changing underlying relationships in the data over time. Traditional machine learning methods assume that the mapping learned from historical data will be valid for new data in the future and the relationships between input and output do not change over time. However, malware is pushed to evolve in order to evade detection, making this assumption false. Moreover, malicious software, as any other piece of software, naturally evolve over time due to changes resulting from adding features, fixing bugs, porting to new environments, etcetera. Therefore, the performance of any ML-based detector naturally degrades over time as malware evolves. To tackle concept drift, Jordaney et al. [69] proposed a framework to identify aging classification models during deployment. This is done by assessing the quality of the prediction made by the machine learning model. It builds and make use of p-values to compute a per-class threshold to identify unreliable predictions. The idea behind is to detect when the model's performance starts to degrade before it actually happens. Contrarily, Pendlebury et al. [70] proposed a set of space and time constraints that eliminate "spatial bias", training and testing distributions not representative of the real-world data, and "temporal bias", incorrect time splits of training and testing sets. In addition, they introduced a set of time-aware performance metrics that allows for the comparison of different classifiers while considering time decay. Following the aforementioned research, new approaches might be developed to deal with concept drift by classification with rejection, in which examples that have drifted far away from the training distribution, and thus are likely to be misclassified by the ML models, are quarantined.

Acknowledgements

This project has received funding from Enterprise Ireland, the Spanish Science and Innovation Ministry funded project PID2019-111544GB-C22, the European Union's Horizon 2020 Research and Innovation Programme under the Marie Skłodowska-Curie grant agreement No 847402. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of CeADAR, University College Dublin, and the University of Lleida.

Conflicts of Interest

The authors declare that they have no known competing financial interests or personal relationships that may appear to influence the work reported in this paper.

References

[1] J. Langevin, J.A. Lewis, Center for Strategic, D.C.) International Studies (Washington, and CSIS Commission on Cybersecurity for the 44th Presidency. *A Human Capital Crisis in Cybersecurity: Technical Proficiency Matters : a White Paper of the CSIS Commission on Cybersecurity for the 44th Presidency*. Center for Strategic and International Studies, 2010.

[2] Mansour Ahmadi, Dmitry Ulyanov, Stanislav Semenov, Mikhail Trofimov, and Giorgio Giacinto. Novel feature extraction, selection and fusion for effective malware family classification. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, CODASPY '16, page 183–194, New York, NY, USA, 2016. Association for Computing Machinery.

https://t.me/learningnets

[3] Y. Zhang, Q. Huang, X. Ma, Z. Yang, and J. Jiang. Using multi-features and ensemble learning method for imbalanced malware classification. In *2016 IEEE Trustcom/BigDataSE/ISPA*, pages 965–973, Aug 2016.

[4] B. N. Narayanan, O. Djaneye-Boundjou, and T. M. Kebede. Performance analysis of machine learning and pattern recognition algorithms for malware classification. In *2016 IEEE National Aerospace and Electronics Conference (NAECON) and Ohio Innovation Summit (OIS)*, pages 338–342, July 2016.

[5] L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath. Malware images: Visualization and automatic classification. In *Proceedings of the 8th International Symposium on Visualization for Cyber Security*, VizSec '11, pages 4:1–4:7, New York, NY, USA, 2011. ACM.

[6] Mitchell Mays, Noah Drabinsky, and Stephan Brandle. Feature selection for malware classification. In *Proceedings of the 28th Modern Artificial Intelligence and Cognitive Science Conference 2017, Fort Wayne, IN, USA, April 28-29, 2017.*, pages 165–170, 2017.

[7] Edward Raff, Richard Zak, R. Cox, J. Sylvester, P. Yacci, R. Ward, Anna Tracy, M. McLean, and C. Nicholas. An investigation of byte n-gram features for malware classification. *Journal of Computer Virology and Hacking Techniques*, 14:1–20, 2016.

[8] D. Gibert, J. Bejar, C. Mateu, J. Planes, D. Solis, and R. Vicens. Convolutional neural networks for classification of malware assembly code. In *International Conference of the Catalan Association for Artificial Intelligence*, pages 221–226, Oct 2017.

[9] Daniel Gibert, Carles Mateu, Jordi Planes, and Joao Marques-Silva. Auditing static machine learning anti-malware tools against metamorphic attacks. *Computers & Security*, 102:102159, 2021.

[10] Daniel Gibert, Carles Mateu, Jordi Planes, and Ramon Vicens. Classification of malware by using structural entropy on convolutional neural networks. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 7759–7764, 2018.

[11] Daniel Gibert, Carles Mateu, Jordi Planes, and Ramon Vicens. Using convolutional neural networks for classification of malware represented as images. *Journal of Computer Virology and Hacking Techniques*, Aug 2018.

[12] Riaz Ullah Khan, Xiaosong Zhang, and Rajesh Kumar. Analysis of resnet and googlenet models for malware detection. *Journal of Computer Virology and Hacking Techniques*, Aug 2018.

[13] Daniel Gibert, Carles Mateu, and Jordi Planes. An end-to-end deep learning architecture for classification of malware's binary content. In *Artificial Neural Networks and Machine Learning - ICANN 2018 - 27th International Conference on Artificial Neural Networks, Rhodes, Greece, October 4-7, 2018, Proceedings, Part III*, pages 383–391, 2018.

[14] M. Yousefi-Azar, V. Varadharajan, L. Hamey, and U. Tupakula. Autoencoder-based feature learning for cyber security applications. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 3854–3861, May 2017.

[15] Daniel Gibert, Carles Mateu, and Jordi Planes. The rise of machine learning for detection and classification of malware: Research developments, trends and challenges. *Journal of Network and Computer Applications*, 153:102526, 2020.

[16] Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29:1189–1232, 2000.

[17] Royi Ronen, Marian Radu, Corina Feuerstein, Elad Yom-Tov, and Mansour Ahmadi. Microsoft malware classification challenge. *CoRR*, abs/1802.10135, 2018.

[18] Daniele Ucci, Leonardo Aniello, and Roberto Baldoni. Survey of machine learning techniques for malware analysis. *Computers & Security*, 81:123–147, 2019.

[19] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles K. Nicholas. Malware detection by eating a whole EXE. In *The Workshops of the The Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, February 2-7, 2018.*, pages 268–276, 2018.

[20] Marek Krčál, Ondřej Švec, Martin Bálek, and Otakar Jašek. Deep convolutional malware classifiers can learn from raw executables and labels only, 2018.

[21] Yongkang Jiang, Shenghong Li, Yue Wu, and Futai Zou. A novel image-based malware classification model using deep learning. In *Neural Information Processing: 26th International Conference, ICONIP 2019, Sydney, NSW, Australia, December 12–15, 2019, Proceedings, Part II*, page 150–161, Berlin, Heidelberg, 2019. Springer-Verlag.

https://t.me/learningnets

[22] Xianwei Gao, Changzhen Hu, Chun Shan, Baoxu Liu, Zequn Niu, and Hui Xie. Malware classification for the cloud via semi-supervised transfer learning. *Journal of Information Security and Applications*, 55:102661, 2020.

[23] Mao Xiao, Chun Guo, Guowei Shen, Yunhe Cui, and Chaohui Jiang. Image-based malware classification using section distribution information. *Computers & Security*, 110:102420, 2021.

[24] Sudhakar and Sushil Kumar. Mcft-cnn: Malware classification with fine-tune convolution neural networks using traditional and transfer learning in internet of things. *Future Generation Computer Systems*, 125:334–351, 2021.

[25] Barath Narayanan Narayanan and Venkata Salini Priyamvada Davuluru. Ensemble malware classification system using deep neural networks. *Electronics*, 9(5), 2020.

[26] Venkata Salini Priyamvada Davuluru, Barath Narayanan Narayanan, and Eric J. Balster. Convolutional neural networks as classification tools and feature extractors for distinguishing malware programs. In *2019 IEEE National Aerospace and Electronics Conference (NAECON)*, pages 273–278, 2019.

[27] R. Lyda and J. Hamrock. Using entropy analysis to find encrypted and packed malware. *IEEE Security & Privacy*, 5(2):40–45, 2007.

[28] L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath. Malware images: Visualization and automatic classification. In *Proceedings of the 8th International Symposium on Visualization for Cyber Security*, VizSec '11, New York, NY, USA, 2011. Association for Computing Machinery.

[29] R. M. Haralick, K. Shanmugam, and I. Dinstein. Textural features for image classification. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-3(6):610–621, 1973.

[30] T. Ojala, M. Pietikainen, and D. Harwood. Performance evaluation of texture measures with classification based on kullback discrimination of distributions. In *Proceedings of 12th International Conference on Pattern Recognition*, volume 1, pages 582–585 vol.1, 1994.

[31] Daniel Bilar. Statistical structures: Fingerprinting malware for classification and analysis. 01 2006.

[32] BXNET. Top maliciously used apis. `https://www.bnxnet.com/top-maliciously-used-apis/`. Offline.

[33] Donabelle Baysa, Richard M. Low, and Mark Stamp. Structural entropy and metamorphic malware. *J. Comput. Virol. Hacking Tech.*, 9(4):179–192, 2013.

[34] Josif Grabocka, Nicolas Schilling, Martin Wistuba, and Lars Schmidt-Thieme. Learning time-series shapelets. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, page 392–401, New York, NY, USA, 2014. Association for Computing Machinery.

[35] Igor Santos, Yoseba K. Penya, Jaime Devesa, and Pablo G. Bringas. N-grams-based file signatures for malware detection. In *in: Proceedings of the 2009 International Conference on Enterprise Information Systems (ICEIS), Volume AIDSS*, pages 317–320.

[36] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. *CoRR*, abs/1603.02754, 2016.

[37] Hyrum S. Anderson and Phil Roth. EMBER: an open dataset for training static PE malware machine learning models. *CoRR*, abs/1804.04637, 2018.

[38] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[39] Karl Pearson F.R.S. X. on the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 50(302):157–175, 1900.

[40] Lars Sthle and Svante Wold. Analysis of variance (anova). *Chemometrics and Intelligent Laboratory Systems*, 6(4):259–272, 1989.

[41] R. Steuer, J. Kurths, C. O. Daub, J. Weise, and J. Selbig. The mutual information: Detecting and evaluating dependencies between variables. *Bioinformatics*, 18:S231–S240, 10 2002.

[42] Temesguen Messay Kebede, Ouboti Djaneye-Boundjou, Barath Narayanan Narayanan, Anca Ralescu, and David Kapp. Classification of malware programs using autoencoders based deep learning architecture and its application to the microsoft malware classification challenge (big 2015) dataset. In *2017 IEEE National Aerospace and Electronics Conference (NAECON)*, pages 70–75, 2017.

https://t.me/learningnets

[43] M. Kalash, M. Rochan, N. Mohammed, N. D. B. Bruce, Y. Wang, and F. Iqbal. Malware classification with deep convolutional neural networks. In *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–5, 2018.

[44] Y. Liu, Y. Lai, Z. Wang, and H. Yan. A new learning approach to malware classification using discriminative feature extraction. *IEEE Access*, 7:13015–13023, 2019.

[45] R. Vinayakumar, M. Alazab, K. P. Soman, P. Poornachandran, and S. Venkatraman. Robust intelligent malware detection using deep learning. *IEEE Access*, 7:46717–46738, 2019.

[46] Wai Weng Lo, Xu Yang, and Yapeng Wang. An xception convolutional neural network for malware classification with transfer learning. In *2019 10th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–5, 2019.

[47] Yanchen Qiao, Qingshan Jiang, Zhenchao Jiang, and Liang Gu. A multi-channel visualization method for malware classification based on deep learning. In *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pages 757–762, 2019.

[48] Aykut Çayır, Uğur Ünal, and Hasan Dağ. Random capsnet forest model for imbalanced malware type classification task. *Computers & Security*, 102:102133, 2021.

[49] Wei-Cheng Lin and Yi-Ren Yeh. Efficient malware classification by binary sequences with one-dimensional convolutional neural networks. *Mathematics*, 10(4), 2022.

[50] Baoguo Yuan, Junfeng Wang, Dong Liu, Wen Guo, Peng Wu, and Xuhua Bao. Byte-level malware classification based on markov images and deep learning. *Computers & Security*, 92:101740, 2020.

[51] Jin-Young Kim, Seok-Jun Bu, and Sung-Bae Cho. Zero-day malware detection using transferred generative adversarial networks based on deep autoencoders. *Information Sciences*, 460-461:83–102, 2018.

[52] Xiaoliang Zhang, Kehe Wu, Zuge Chen, and Chenyi Zhang. Malcaps: A capsule network based model for the malware classification. *Processes*, 9(6), 2021.

[53] Guoqing Xiao, Jingning Li, Yuedan Chen, and Kenli Li. Malfcs: An effective malware classification framework with automated feature extraction based on deep convolutional neural networks. *Journal of Parallel and Distributed Computing*, 141:49–58, 2020.

[54] Jiaqi Yan, Guanhua Yan, and Dong Jin. Classifying malware represented as control flow graphs using deep graph convolutional neural network. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 52–63, 2019.

[55] X. Hu, J. Jang, T. Wang, Z. Ashraf, M. Ph. Stoecklin, and D. Kirat. Scalable malware classification with multifaceted content features and threat intelligence. *IBM J. Res. Dev.*, 60(4):6:1–6:11, July 2016.

[56] Daniel Gibert, Carles Mateu, and Jordi Planes. A hierarchical convolutional neural network for malware classification. In *International Joint Conference on Neural Networks, IJCNN 2019 Budapest, Hungary, July 14-19, 2019*, pages 1–8. IEEE, 2019.

[57] Niall McLaughlin, Jesus Martinez del Rincon, BooJoong Kang, Suleiman Yerima, Paul Miller, Sakir Sezer, Yeganeh Safaei, Erik Trickel, Ziming Zhao, Adam Doupé, and Gail Joon Ahn. Deep android malware detection. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, CODASPY '17, page 301–308, New York, NY, USA, 2017. Association for Computing Machinery.

[58] Jake Drew, Michael Hahsler, and Tyler Moore. Polymorphic malware detection using sequence classification methods and ensembles. *EURASIP J. Inf. Secur.*, 2017(1), dec 2017.

[59] Jin-Young Kim and Sung-Bae Cho. Obfuscated malware detection using deep generative model based on global/local features. *Computers & Security*, 112:102501, 2022.

[60] Quan Le, Oisín Boydell, Brian Mac Namee, and Mark Scanlon. Deep learning at the shallow end: Malware classification for non-domain experts. *Digital Investigation*, 26:S118–S126, 2018.

[61] Temesguen Messay-Kebede, Barath Narayanan Narayanan, and Ouboti Djaneye-Boundjou. Combination of traditional and deep learning based architectures to overcome class imbalance and its application to malware classification. In *NAECON 2018 - IEEE National Aerospace and Electronics Conference*, pages 73–77, 2018.

[62] D. Gibert, C. Mateu, and J. Planes. Orthrus: A bimodal learning architecture for malware classification. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2020.

[63] Daniel Gibert, Carles Mateu, and Jordi Planes. Hydra: A multimodal deep learning framework for malware classification. *Computers & Security*, 95:101873, 2020.

https://t.me/learningnets

[64] Weiwei Hu and Ying Tan. Generating adversarial malware examples for black-box attacks based on GAN. *CoRR*, abs/1702.05983, 2017.

[65] Octavian Suciu, Scott E. Coull, and Jeffrey Johns. Exploring adversarial examples in malware detection. In *2019 IEEE Security and Privacy Workshops (SPW)*, pages 8–14, 2019.

[66] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[67] Luca Demetrio, Battista Biggio, Giovanni Lagorio, Fabio Roli, and Alessandro Armando. Functionality-preserving black-box optimization of adversarial windows malware. *IEEE Transactions on Information Forensics and Security*, 16:3469–3478, 2021.

[68] Luca Demetrio, Scott E. Coull, Battista Biggio, Giovanni Lagorio, Alessandro Armando, and Fabio Roli. Adversarial exemples: A survey and experimental evaluation of practical attacks on machine learning for windows malware detection. *ACM Trans. Priv. Secur.*, 24(4), September 2021.

[69] Roberto Jordaney, Kumar Sharad, Santanu K. Dash, Zhi Wang, Davide Papini, Ilia Nouretdinov, and Lorenzo Cavallaro. Transcend: Detecting concept drift in malware classification models. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 625–642, Vancouver, BC, August 2017. USENIX Association.

[70] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. TESSERACT: Eliminating experimental bias in malware classification across space and time. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 729–746, Santa Clara, CA, August 2019. USENIX Association.

A Feature Importance by Category

Following, the readers will find the top 20 features of each feature category. The importance of each feature is defined as the average gain across all splits the feature has been used in.

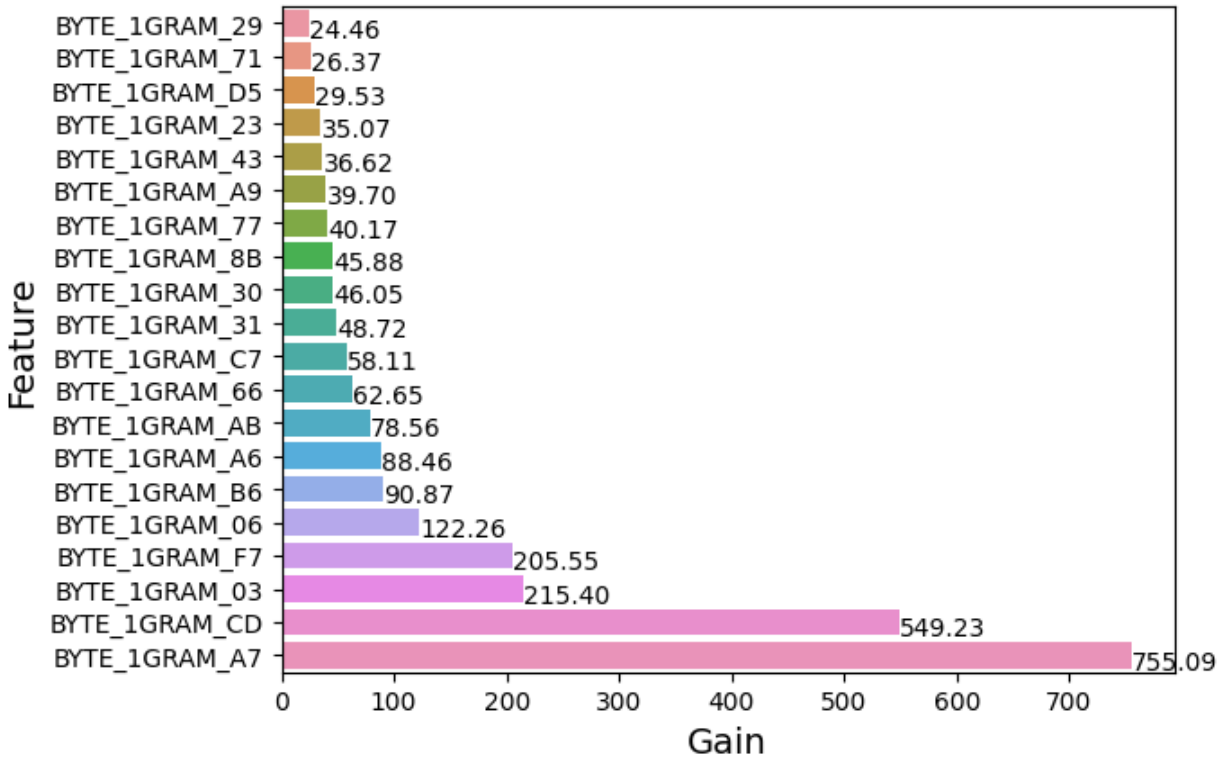


Figure 12: Top 20 *BYTE_1G* features.

https://t.me/learningnets

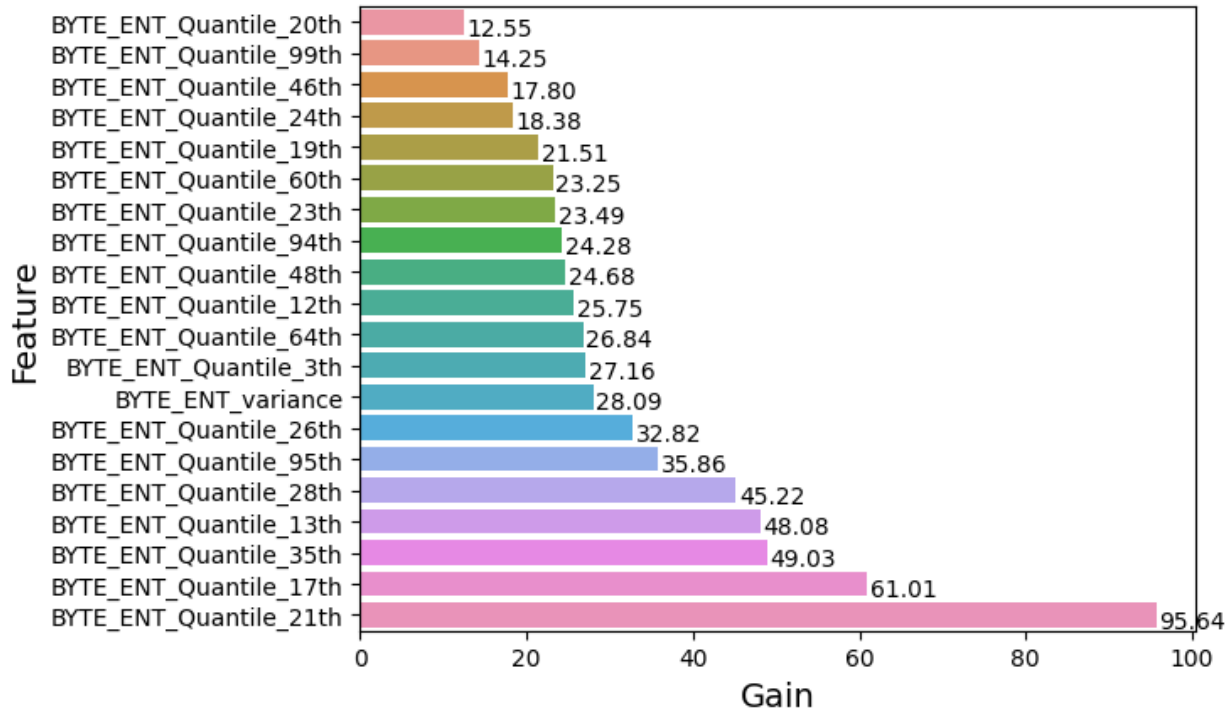


Figure 13: Top 20 *BYTE_ENT* features.

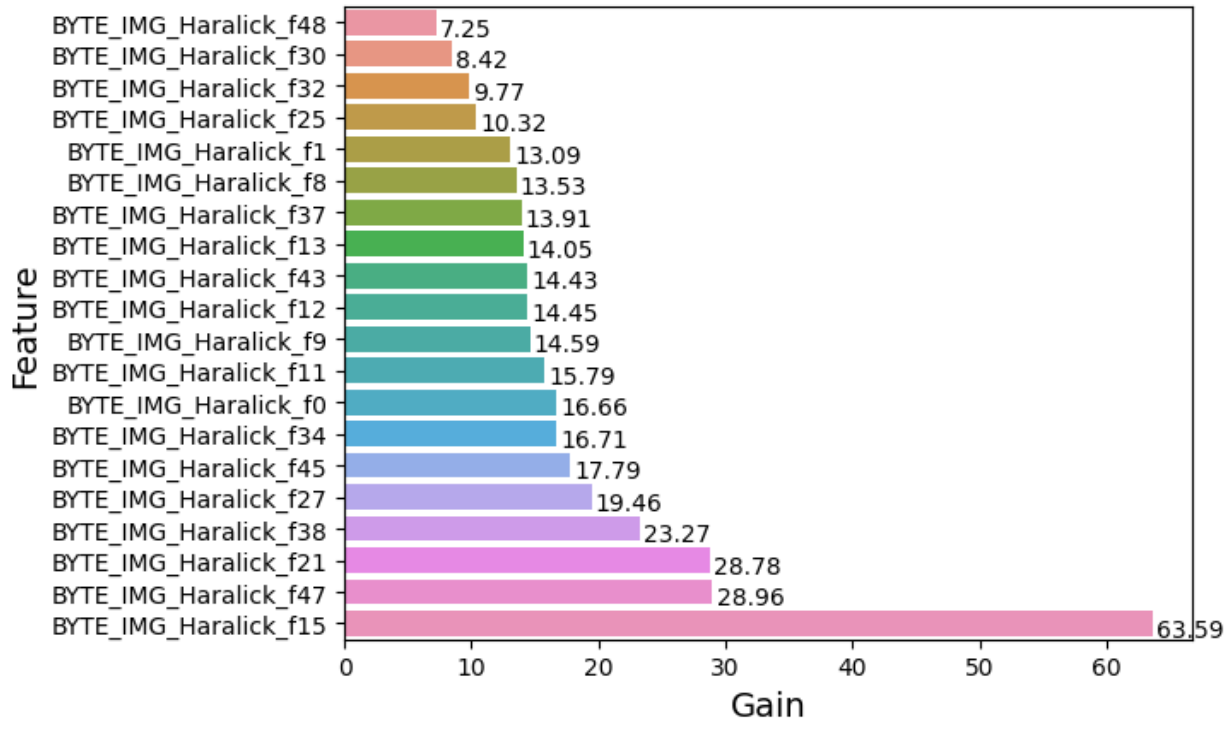


Figure 14: Top 20 *BYTE_IMG_HAR* features.

https://t.me/learningnets

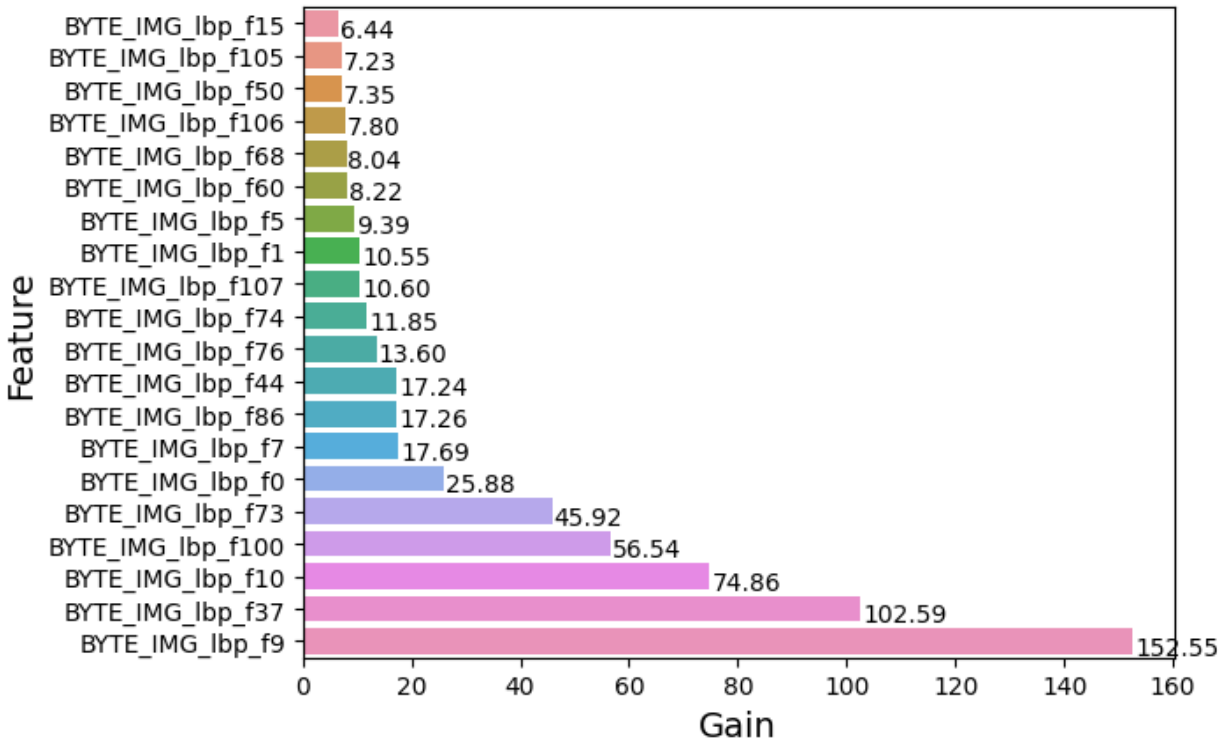


Figure 15: Top 20 *BYTE_IMG_LBP* features.

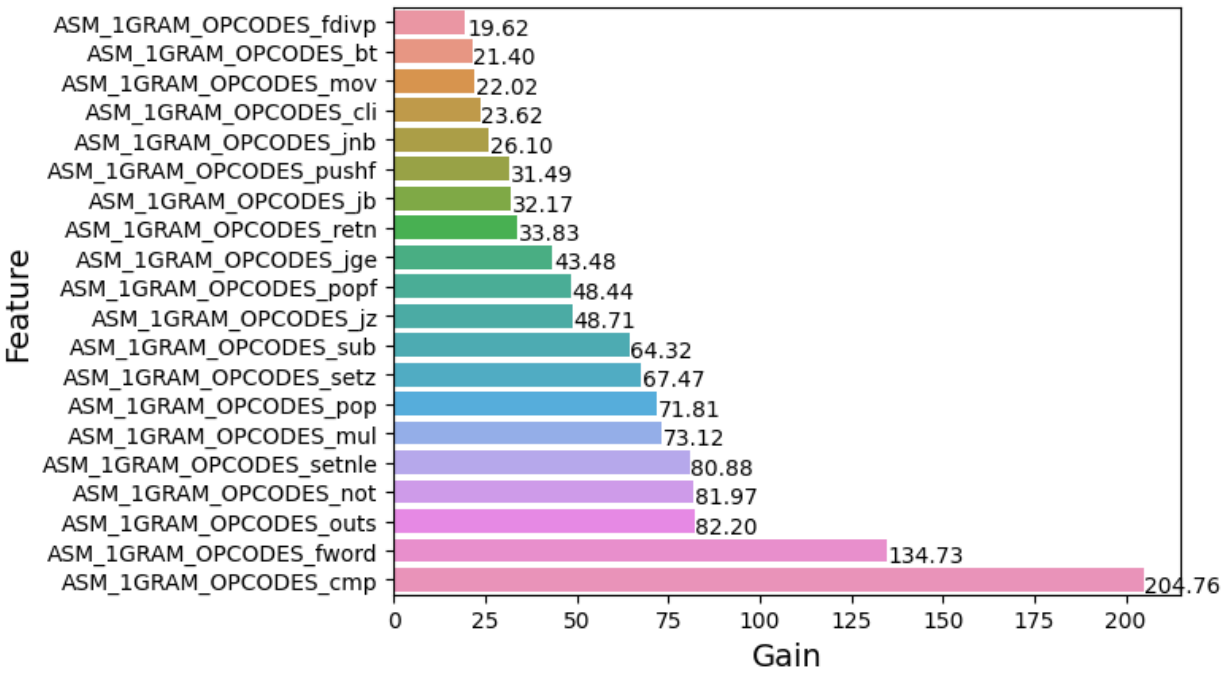


Figure 16: Top 20 *ASM_OPC* features.

https://t.me/learningnets

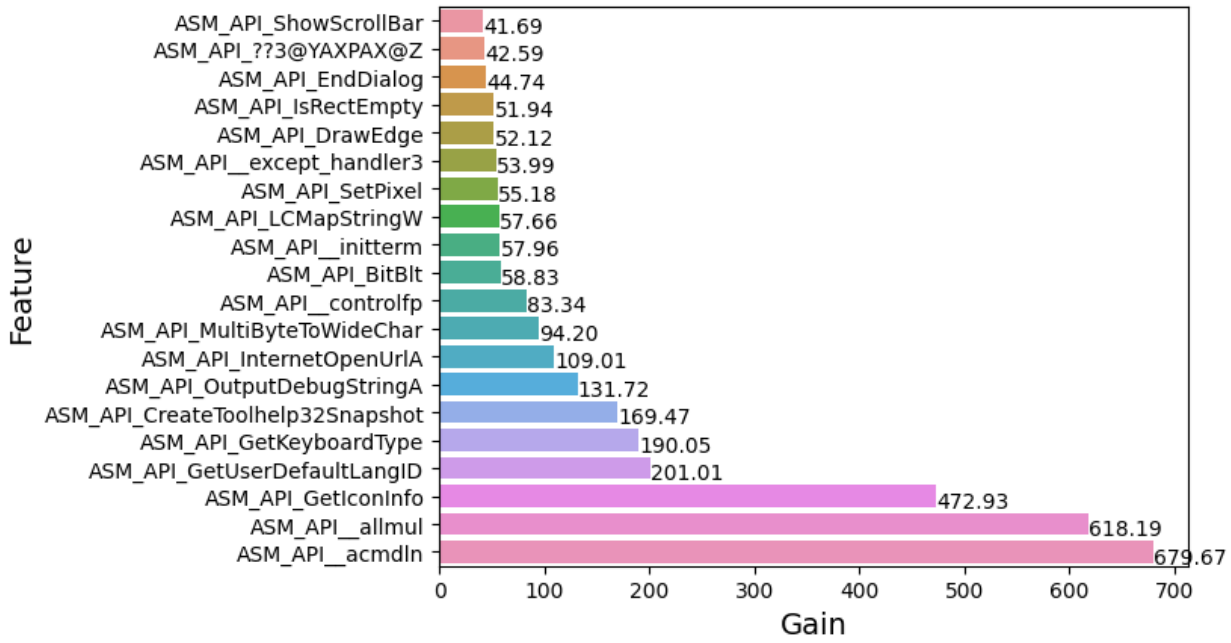


Figure 17: Top 20 *ASM_API* features.

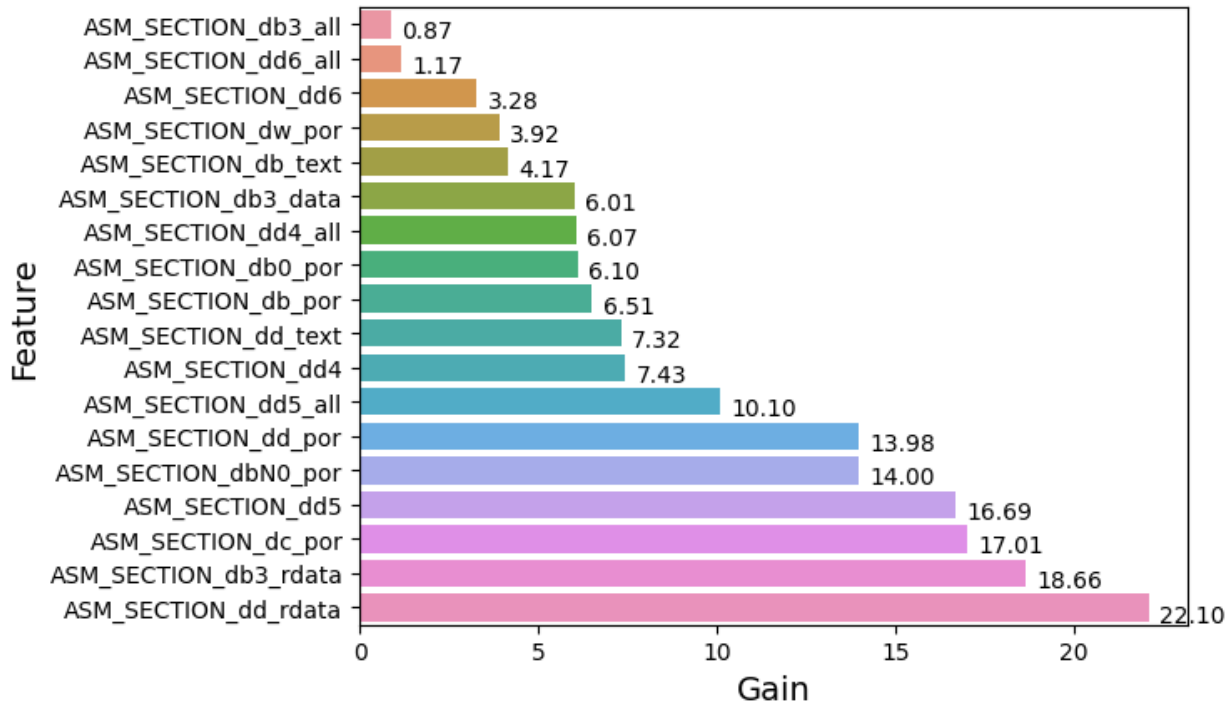


Figure 18: Top 20 *ASM_DD* features.

https://t.me/learningnets

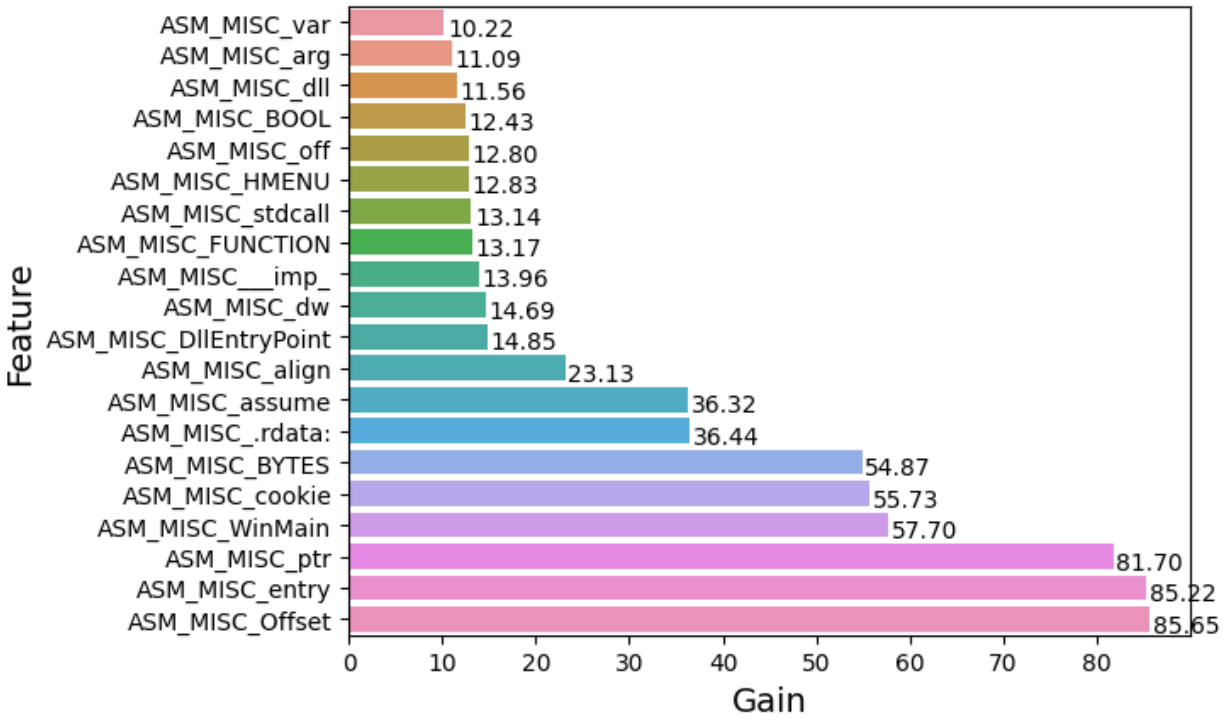


Figure 19: Top 20 *ASM_MISC* features.

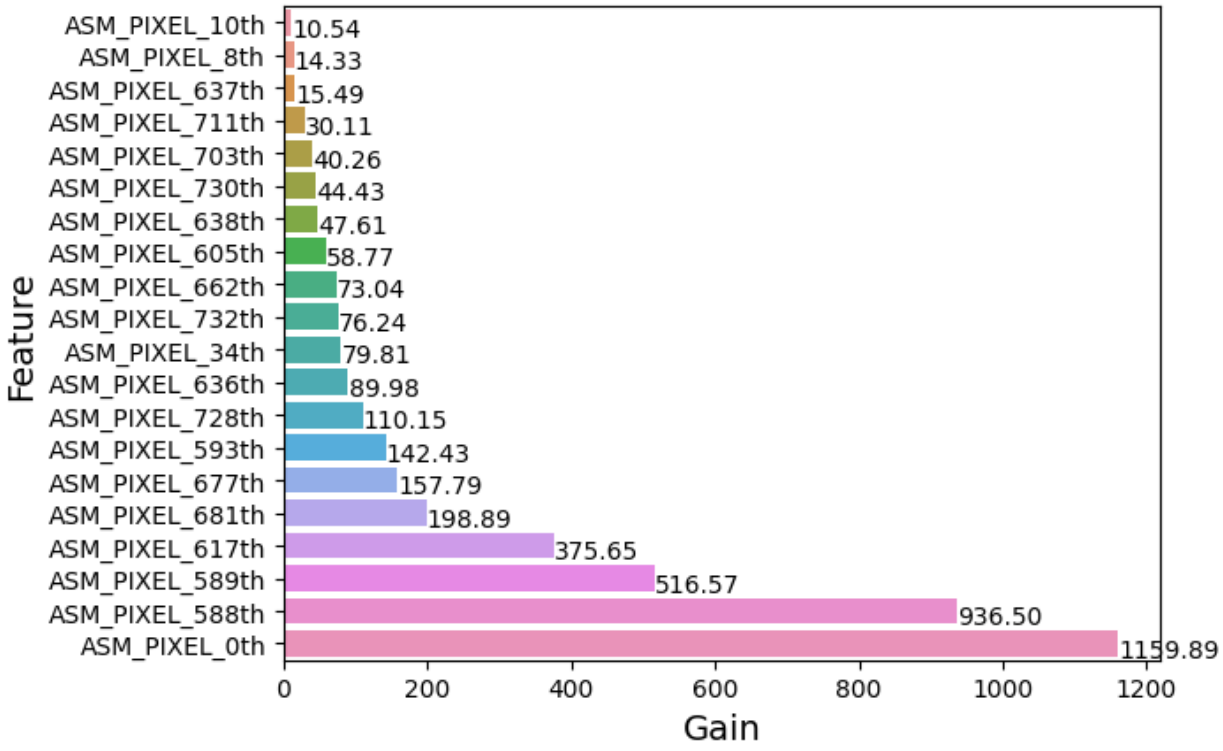


Figure 20: Top 20 *ASM_PIXEL* features.

https://t.me/learningnets

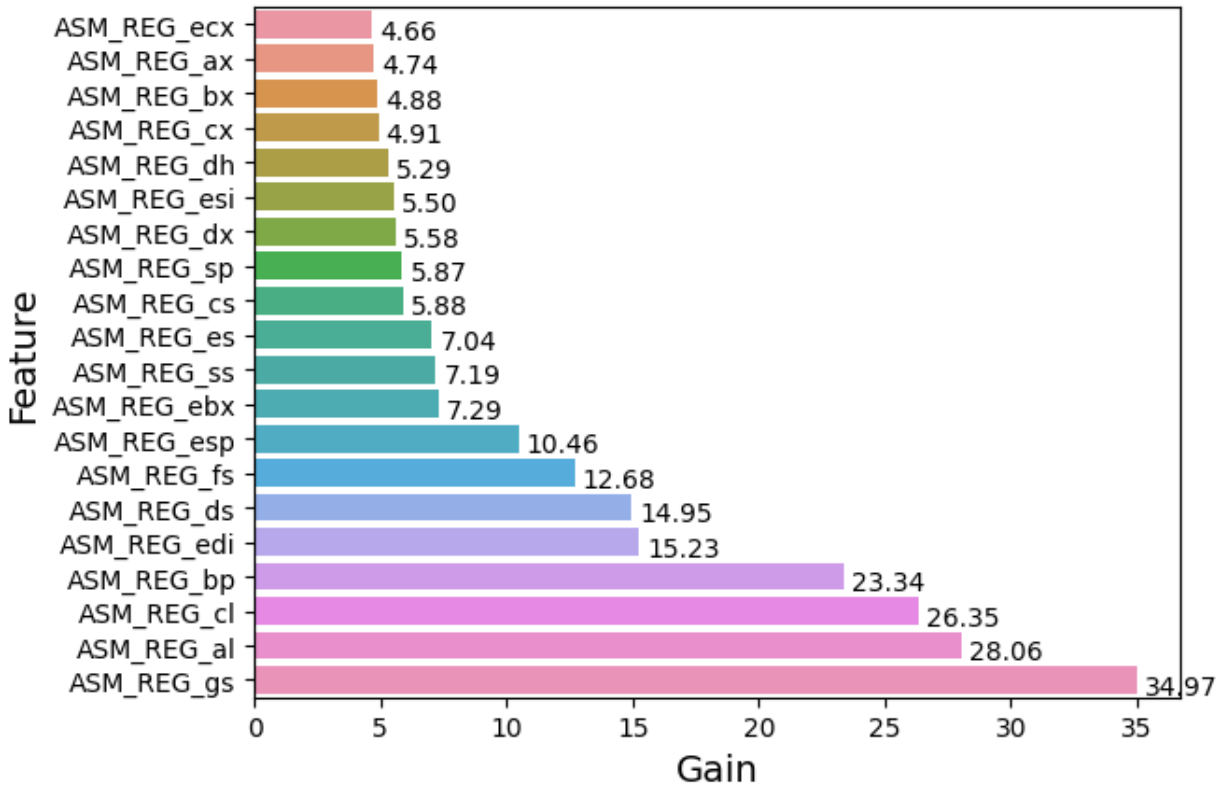


Figure 21: Top 20 *ASM_REG* features.

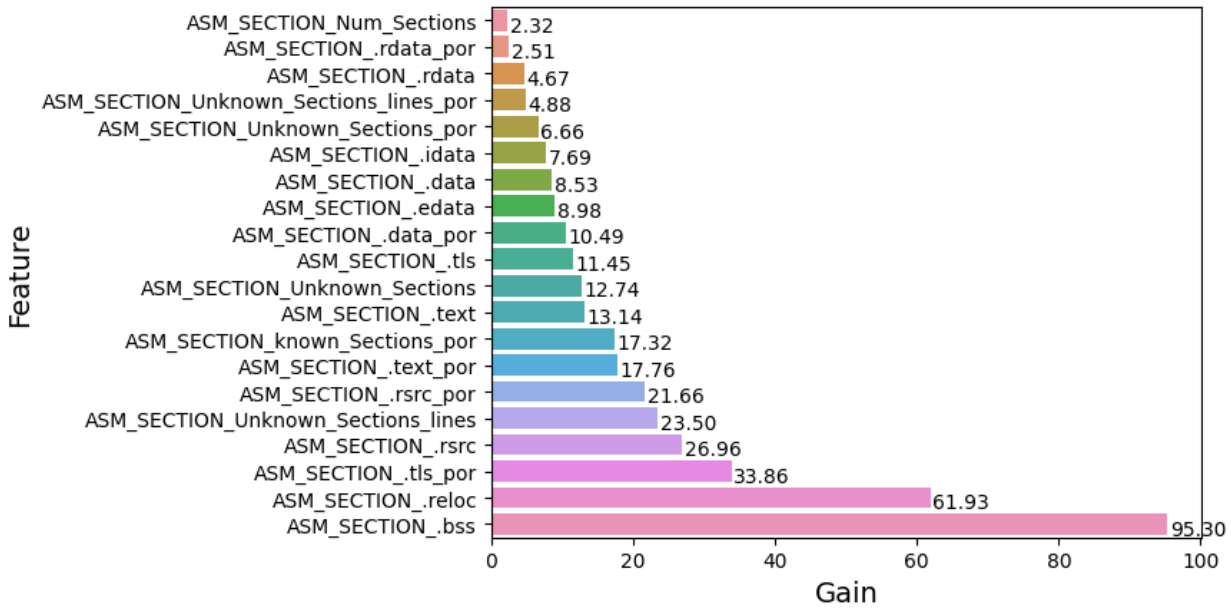


Figure 22: Top 20 *ASM_SEC* features.

https://t.me/learningnets

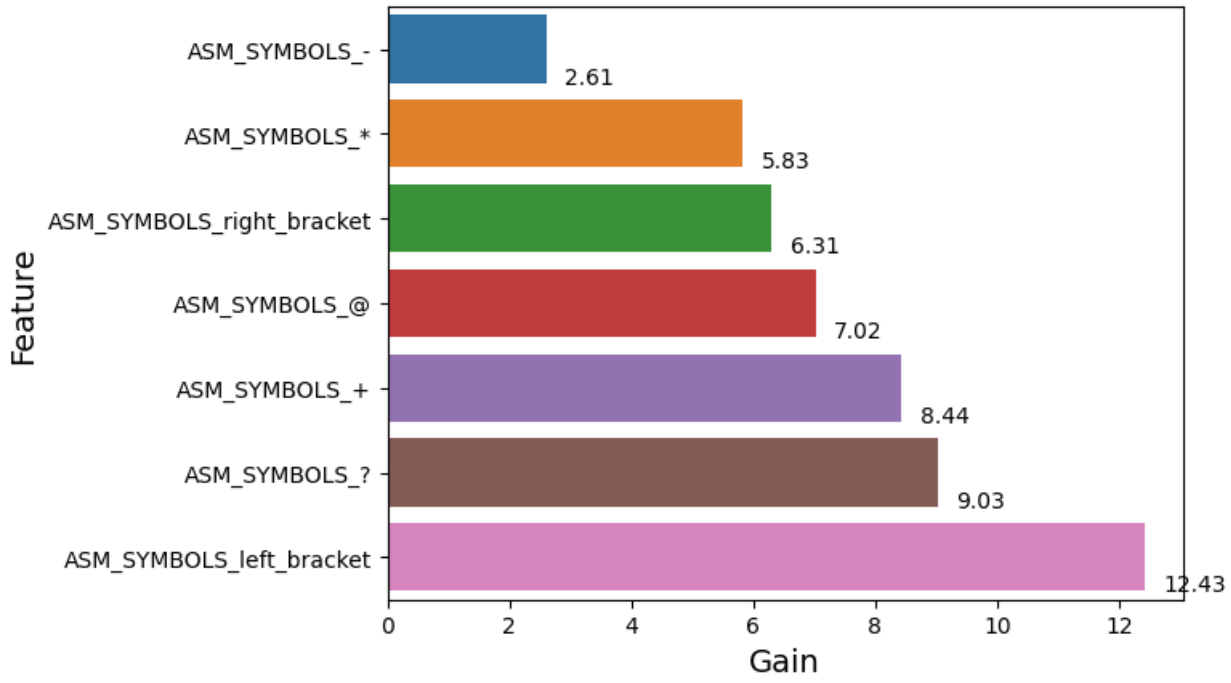


Figure 23: *ASM_SYM* feature importance.

https://t.me/learningnets