

TECHNICAL UNIVERSITY OF CRETE  
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING  
TELECOMMUNICATIONS DIVISION



# Malware Detection using Machine Learning: A double input architecture

by

Panagiotis Bellonias

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF  
THE REQUIREMENTS FOR THE DIPLOMA OF  
ELECTRICAL AND COMPUTER ENGINEERING

July 2020

## THESIS COMMITTEE

Professor Aggelos Bletsas, *Thesis Supervisor*  
Associate Professor Michail G. Lagoudakis  
Professor Vasilis Katos, Bournemouth University, UK

# Abstract

This thesis will try to combine two different applications, data classification, and image classification to further improve malware detection. The effectiveness of a double input architecture model is evaluated. The neural network developed takes two different kinds of inputs, the grayscale image representation of the sample and features extracted from the headers of the file. For this purpose, a dataset has been created containing data from sources like MalShare [1] and VirusTotal [2]. Feature encoding has been used for creating a mathematical summary of the features and standardizing the input vector. The implemented model is compared to two different neural networks to highlight its effectiveness. The first one uses an image representation of the executable file as an input and the second one uses only features from the headers of the file. The double input architecture proved to outperform its contestants with an area under receiver operating characteristic (ROC) curve (AUC) equal to 0.989. Furthermore, state-of-the-art antivirus products were compared to the proposed architecture, even though the latter was trained with a relatively limited dataset. The proposed neural network was placed third with a True Positive Rate of 0.972. Complete sources are provided for reproducing the proposed model and the derived results [3]. The importance of large dataset availability in such domains should not be overlooked.

Thesis Supervisor: Professor Aggelos Bletsas

# Acknowledgements

Many thanks to my supervisor, Professor Aggelos Bletsas, for providing guidance and feedback throughout this project but mostly for his patience and understanding. Thanks also to my family, for motivating me through this journey, and for providing guidance and a sounding board when required. Lastly, I would like to thank Professor Vasileios Katos for being my advisor outside Technical University of Crete, and for giving the opportunity to develop in the cyber security field.

# Table of Contents

<b>Table of Contents</b> . . . . .	4
<b>List of Figures</b> . . . . .	6
<b>List of Tables</b> . . . . .	7
<b>1 Introduction</b> . . . . .	8
1.1 Objective . . . . .	9
1.2 Outline . . . . .	10
<b>2 Literature Review</b> . . . . .	11
2.1 Static Malware Analysis . . . . .	11
2.1.1 Signature Evasion . . . . .	11
2.1.2 Code Obfuscation . . . . .	12
2.1.3 Malware Packing . . . . .	13
2.2 Machine Learning for Malware Detection . . . . .	14
2.2.1 N-Grams of Bytes Sequences . . . . .	15
2.2.2 N-Grams of Opcode sequences . . . . .	15
2.2.3 API Calls . . . . .	16
2.2.4 Headers Meta Data . . . . .	17
2.2.5 Image Representation of Malware . . . . .	17
<b>3 Background</b> . . . . .	20
3.1 Portable Executable Format . . . . .	20
3.1.1 MS-DOS Stub . . . . .	20
3.1.2 COFF Header . . . . .	22
3.1.3 Optional Header . . . . .	22

---

3.1.4	Section Header . . . . .	23
3.2	Artificial Neural Networks . . . . .	24
3.3	Convolutional Neural Networks . . . . .	28
3.3.1	Convolution Layer . . . . .	29
3.3.2	Non-linear Layer . . . . .	31
3.3.3	Pooling Layer . . . . .	32
3.3.4	Fully Connected Layer . . . . .	33
<b>4</b>	<b>Implementation . . . . .</b>	<b>34</b>
4.1	Data Collection . . . . .	34
4.2	Lab Enviroment . . . . .	35
4.3	Extracted Features and Dataset . . . . .	35
4.4	Samples to Images . . . . .	39
4.5	Classifiers . . . . .	40
<b>5</b>	<b>Experiments and Results . . . . .</b>	<b>44</b>
5.1	Experimental Setup . . . . .	44
5.2	Metrics . . . . .	44
5.3	Results . . . . .	45
5.3.1	CNN Classifier . . . . .	46
5.3.2	DNN Classifier . . . . .	48
5.3.3	Enhanced Classifier . . . . .	49
5.3.4	Malware in the wild . . . . .	50
5.3.5	Overall Results . . . . .	50
<b>6</b>	<b>Conclusion . . . . .</b>	<b>52</b>
	<b>Bibliography . . . . .</b>	<b>53</b>

# List of Figures

2.1	The images in the first row are images of 3 instances of malware belonging to the family Fakerean [4] and those in the second row belong to the family Dontovo.A [5]. . . . .	18
3.1	PE File Format [6]. . . . .	21
3.2	Basic structure of a neuron. . . . .	25
3.3	Sigmoid function and Hyperbolic tangent function. . . . .	25
3.4	Basic structure of multi-layer perceptron. . . . .	26
3.5	Example of a convolutional neural network. . . . .	29
3.6	Example of kernel calculation within convolution layer. . . . .	30
3.7	Example of rectified linear unit transformation. . . . .	32
3.8	Two classic methods for pooling. . . . .	33
4.1	Different properties along samples. . . . .	37
4.2	Distribution of byte values of the the entry point. . . . .	37
4.3	Ratio of virtual size and disk size. . . . .	38
4.4	Summary of the Convolutional Neural Network. . . . .	41
4.5	Summary of the Dense Neural Network. . . . .	42
4.6	Summary of the Enhanced Neural Network. . . . .	43
5.1	CNN Results. . . . .	47
5.2	DNN Results. . . . .	48
5.3	Enhanced Classifier Results. . . . .	49
5.4	ROC Curve results. . . . .	51

# List of Tables

3.1	COFF header. . . . .	22
3.2	Optional header parts. . . . .	22
3.3	Optional header standard fields. . . . .	23
4.1	Samples collected. . . . .	35
4.2	Virtual Machine System Settings. . . . .	35
4.3	Important properties of a file. . . . .	36
4.4	Dataset structure. . . . .	39
5.1	Results in unknown samples. . . . .	50

# Chapter 1

## Introduction

Malware is one of the most serious security threats and spreads autonomously through vulnerabilities or careless users. To prevent infection or remove malware from a computer system, it is of utmost importance to detect malware successfully. The concept of malware detection mainly deals with the analysis of executable files to establish malicious intent. As anti-malware software develops, malicious executables are becoming more and more sophisticated. Thus, research has been headed into the development of more advanced detection techniques. Two main analysis techniques are used to detect the maliciousness of a portable executable: static, and dynamic analysis.

**Static malware analysis** refers to analyzing the binary file without execution. It is used to confirm, whether the file being inspected is malicious or not. It is the easiest to perform and allows extraction of the metadata associated with the target binary, such as functions and libraries being called by the executable. Static analysis acts as a stepping stone as it can often provide interesting information that will determine where to focus on the next steps of the analysis.

**Dynamic malware analysis**, unlike static malware analysis, is performed by observing the behavior of the malware, while running its code in a controlled environment. This technique reveals valuable insights into the activity of the binary during its execution. The target file may also be debugged, while running using a debugger, such as GNU Debugger (GDB) [7], to watch the behavior of the malware, analyze system calls or other patterns, while its code is being executed. This technique requires considerable resources and can be evaded in various ways. For the purpose of this thesis, we will focus on static malware detection.

During the last decade, machine learning has triggered a radical shift in

---

many sectors, including cybersecurity. There is a general belief among cybersecurity experts that AI-powered antimalware tools will help detect modern malware attacks and improve scanning engines. Neural networks are used today for a variety of applications like data classification, data prediction, image recognition, natural language processing, and so on.

This thesis will try to combine two different applications, data classification, and image classification to further improve malware detection. Machine learning is split into two main categories: supervised, and unsupervised learning.

**Supervised learning** is the process of teaching a model by feeding it input data as well as correct output data. This input/output pair is usually referred to as "labeled data". Think of a teacher who, knowing the correct answer, will either reward marks to or take marks from a student based on the correctness of her response to a question. Supervised learning is often used to create machine learning models for two types of problems, regression and classification.

Regression is a technique that aims to reproduce the output value. It can be used, for example, to predict the price of some product, like a price of a house in a specific city or the value of a stock.

Classification is a technique that aims to reproduce class assignments. It can predict the response value and the data is separated into "classes". Malware detection is a classification problem.

**Unsupervised learning** represents a subset of machine learning tasks that are based around using unlabeled training data, which is data that does not have any kind of label designating its classification. Compared to supervised learning, where training data is labeled with the appropriate classifications, methods using unsupervised learning must learn relationships between elements in a dataset without data labeling.

## 1.1 Objective

The objective of this thesis is to design and evaluate a neural network for portable executable files to classify them as malicious or benign, based on

supervised learning. The model developed takes two different kinds of inputs, a grayscale image representation and features extracted from the headers of the file. For this purpose, a dataset has been created containing data from sources like MalShare and VirusTotal. Feature encoding has been used for creating a mathematical summary of the features and standardizing the input vector.

The model developed is compared to two different neural networks to highlight its effectiveness. The first one uses an image representation of the executable file as an input and the second one uses only features from the headers of the file. Furthermore, state-of-art antivirus products were compared to the architecture mentioned. Complete sources are provided for reproducing the proposed model and the derived results.

## 1.2 Outline

- **Chapter 1** introduces the concepts covered in this thesis.
- **Chapter 2** presents previous research in the field of static malware analysis with machine learning.
- **Chapter 3** defines the format of the portable executable file and analyzes its headers, which need to be studied to better understand the implemented model. Moreover, the necessary background regarding the main deep learning algorithms is described.
- **Chapter 4** presents in detail the steps and processes involved in creating the dataset as well as implementing the models.
- **Chapter 5** reviews the experimental results and their implications in the real world.
- **Chapter 6** summarizes the results, the model, and areas of research that have not been covered in this thesis. It also talks about potential gaps in this thesis and future research.

# Chapter 2

## Literature Review

This section covers work published in using machine learning for malware detection. The basic models developed during this thesis are similar to the implementations described in this section. However, most of the literature is not reproducible due to the lack of availability of the data set used, or the use of proprietary frameworks for obtaining results. Related work in this field is also covered which deals with malware detection on other platforms using static as well as dynamic analysis of files.

### 2.1 Static Malware Analysis

This technique refers to analyzing the Portable Executable files (PE files) without running them. There is a variety of challenges lying in this approach, most of which are solved by dynamic analysis. The most popular challenges are presented below, along with the incapability of semantic-analyzers in addressing them.

#### 2.1.1 Signature Evasion

Conventional malware detection products work by examining each object and calculating its digital signature. Signature-based detection is an anti-malware approach that identifies the presence of a malware infection or instance by matching at least one byte code pattern of the software in question with the database of signatures of known malicious programs [8], also known as blacklists.

This method requires a database of signatures continuously updated. To maintain the database, some experts analyse every new malicious programs and try to reverse engineer a corresponding signature, with the constraint to

---

produce less possible false-positives. Bonfante *et al.* [9] proposed a strategy based on control flow graphs as signatures to perform detection to combat this problem. They designed a graph, which consisted of nodes for all commonly used assembly instructions. A reduced version of the graph was used as a signature to detect malicious samples. According to their experiments, that strategy resulted in better overall detection accuracy for larger samples.

Unfortunately, today's advanced malware can alter its signature to avoid detection. Signatures are created by examining the internal components of an object. Skilled malware authors modify these components while preserving the object's functionality. There are multiple transformation techniques used by malware authors, such as code permutation, register renaming, expanding and shrinking code, insertion of garbage code or other constructs, which can alter a signature. Another thing to keep into consideration is that advanced malware is often designed to be single-use, targeting just one organization or a few people within one organization. This narrow focus greatly reduces the odds that its signature will ever appear in a database of malicious objects.

### 2.1.2 Code Obfuscation

Obfuscated programs are ones whose execution is hidden by malicious actors. Several techniques include dead code insertion, register reassignment, subroutine reordering, instruction substitution, and code manipulation.

**Dead-code insertion** adds some NOP (No operation Performed) instructions or inserts ineffective PUSH/ POP statements to a program to change its look, but keep its same behavior.

**Register reassignment** works by switching registers or by reassigning the value of one register to unused one. For example, EAX is reassigned to EBX register. EAX is used in arithmetic operations and EBX points to the address space containing initialized static variables.

**Subroutine reordering** is a group of program operations that do a specific task. This technique changes the subroutines order randomly in the program.

In **instruction substitution**, original instructions that perform the

---

same function are replaced by equivalent ones, such as replacing MOV instruction with PUSH instruction.

In **code integration**, malware embeds itself to another legal program. To apply this technique, malware decompiles its targeted program and adds itself in between its source code [10]. Code integration is considered as one of the most sophisticated obfuscation techniques that allows malware to evade detection.

Mosel *et al.* [11] highlighted a significant flaw in static malware analysis techniques, simply by using opaque constants to obscure program control flow. The semantic analysis was beaten by introducing a randomized approach to calculating constants in real-time. One such method mentioned is to use a random seed to generate addresses where variables are stored, or to daisy-chain the process and store variables in addresses present in other addresses. However, calculating the value of certain constants is considered an NP-hard problem. In that paper, the use of the 3SAT algorithm is discussed, which is difficult to be computed in polynomial time.

Preda *et al.* [12] proposed a semantics-based approach to compare the similarity between original malicious code and obfuscated malware code. Research showed that by adding NP-hard computation or similar methods, obfuscation techniques like NOP insertion, command substitution, and variable renaming could be detected successfully. However, the practical implementation of this approach has not been fully realized.

### 2.1.3 Malware Packing

Packed programs are a subset of obfuscated programs in which the malicious program is compressed and cannot be analyzed [13]. To identify if malware is packed or not, a security professional can carry a static check on it and if an extremely small number of strings is found then there is a near one hundred percent chance that the code is malicious. Different types of encryption could be used in combination with such techniques to prevent malware detection. Polypack is a tool developed to highlight the fact that packers are an effective method of evading anti-virus and anti-malware software [14]. Polypack uses

---

an array of packers and antivirus engines as a feedback mechanism to select the packer that will result in the optimal evasion of the antivirus engines. Towards understanding the utility and efficacy of such a service, a version of PolyPack which employs 10 packers and 10 popular antivirus engines was developed. Results indicated that the tool provides 2.58 times more effective evasion of antivirus engines than using an average packer.

## 2.2 Machine Learning for Malware Detection

There is a general belief within the cybersecurity industry that AI-powered anti-malware tools will help detect modern malware attacks and improve scanning engines. The number of studies published in the last decade on malware detection techniques that leverage machine learning enhances this belief. According to Google Scholar, the number of research papers published in 2019 is 5130, a 240% increase compared to 2015 and a 925% increase with respect to 2010.

Traditional machine learning approaches can be categorized into two primary groups, static and dynamic approaches, depending on the type of analysis. The main difference between them is that static approaches extract features from the static analysis of malware, while dynamic approaches extract features from the dynamic analysis. Since the work presented here focuses on the static approach, the corresponding literature is being presented.

Static features are extracted from a program without involving its execution. This is important since, in real-life scenarios, the user should not run the suspected executable to detect its maliciousness. In Windows Portable Executable files, static features are derived from two sources of information, the binary content of the executable or the assembly language source file obtained after decompiling and disassembling the binary executable.

### 2.2.1 N-Grams of Bytes Sequences

One of the most common type of features for malware detection is n-grams. An n-gram is a contiguous sequence of n items from a given sequence of text. N-grams can be extracted from the bytes sequences representing the malware's binary content and from the assembly language source code. Many tools have been developed for this purpose such as Hex dump [15].

Masud *et al.* [16] created 4-grams from byte sequences of PE32 files. The features were collected by sliding window of n bytes. This resulted in 200 millions of features using 10-grams for about two thousands files in overall. Furthermore, feature selection was applied to select 500 most valuable features based on Information Gain metric. Information Gain measures the quality of a split during the training of Decision Trees. Achieved accuracy on malware detection was up to 97% using such features.

Similar approach was proposed by Fuyong *et al.* [17]. They calculated the information gain of each bytes n-gram in the training samples and selected K n-grams with the maximum information gain as features. Afterwards, they calculated the averages of each attribute of the feature vectors from the malware and benign samples separately. Lastly, a new piece of software was assigned to one of the two categories according to the similarity between the feature vector of the unknown sample and the average vectors of the two categories.

Another work on byte n-grams described a method to extract bytes n-gram features, with n ranging from 1 to 8, from known malicious samples to assist in classification of unknown executables [18]. As the number of unique n-grams is extremely large, they used a technique called classwise document frequency to reduce the feature space. Finally, different N-gram models were prepared using various classifiers like Naïve Bayes, Instance-based Learner, Decision Trees, Adaboost and Random Forests.

### 2.2.2 N-Grams of Opcode sequences

Opcode sequences or operation codes are set of consecutive low level machine abstractions used to perform various CPU operations. As it was shown [19],

such features can be used to train Machine Learning methods for successful classification of the malware samples. However, there should be a balance between the size of the feature set and the length of n-gram opcode sequence. N-grams with the size of 4 and 5 result in highest classification accuracy as unknown malware samples could be unveiled on a collection of 17,000 malware and 1,000 benign files with a classification accuracy up to 94% [20].

Shabtai *et al.* [21] proposed a framework for detecting malware based on opcode n-gram features with n ranging from 1 to 6. They performed a wide set of experiments to: identify the best term representation, whether it is the Term Frequency or Term Frequency-Inverse Document Frequency, determine the n-gram size, find the optimal K top n-grams and feature selection method, and evaluate the performance of various machine learning algorithms.

Despite their success at detecting malware, n-gram approaches have some issues that are worth mentioning. Researchers have concluded that byte n-grams appear to be learning mostly from string content in an executable, in particular items from the PE header [22]. As there are millions of potential n-grams, for a larger n, feature selection techniques tend to select them as features by frequency of occurrence. This encourages the selection of low entropy features consisting mostly of strings and padding. Also, regardless of what kind of n-grams are learned, an exact match must be obtained when classifying a new sample. Based on the power-law observation of the n-gram distribution, previously unobserved n-grams will be exported from each file. This is a scenario where more features are produced than samples, which is a source of over-fitting.

### 2.2.3 API Calls

API calls are the function calls used by a program to execute specific functionality. There is a distinction between System API calls that are available through standard system DLLs and User API calls provided by user installed software. These are designed to perform a pre-defined task during invocation. Suspicious API calls, anti-VM and anti-debugger hooks and calls can be extracted by PE analysers such as PEframe [23]. Researchers studied 23

malware samples and found that some of the API calls are present only in malicious samples rather than benign software [24]. Function calls can be composed in graphs to represent PE32 header features as nodes, edges and subgraphs [25]. This work shows that ML methods achieve accuracy of 96% on 24 features extracted after analysis of 1,037 malware and 2,072 benign executables. Further, in [26] 20,682 API calls were extracted using PE parser for 1,593 malicious and benign samples. Such large number of extracted features can help create a linearly separable model that is crucial for many ML methods like Support Vector Machines or single-layer Neural Networks.

#### 2.2.4 Headers Meta Data

PE header represents a collection of meta data related to a Portable Executable file. Basic features that can be extracted from a PE32 header are Size of Header, Size of Uninitialized Data, Size of Stack Reserve, which may indicate if a binary file is malicious or benign. Shugang Tang [27] utilized Decision Trees to analyse PE header structural information for describing malicious and benign files. Another academic work used 125 raw header characteristics, 31 section characteristics, 29 section characteristics to detect unknown malware in a semi-supervised approach [28]. T. Wang *et al.* [29] used a dataset containing 7,863 malware samples from VX Heaven web site [30] in addition to 1,908 benign files to develop a SVM based malware detection model with an accuracy of 98%.

#### 2.2.5 Image Representation of Malware

A ground approach for malware visualization was first introduced by Nataraj *et al.* [31] who visualized the malware's binary content as a gray scale image. This is achieved by interpreting every byte as one pixel in an image, where values range from 0 to 255 (0:black, 255:white). Afterwards, the resulting array is reorganized as a 2-D array.

Fig. 2.1 depicts samples from two malware families represented as gray scale images. It is clear that the image representation of samples of a given family is quite similar while distinct from that belonging to a different family.

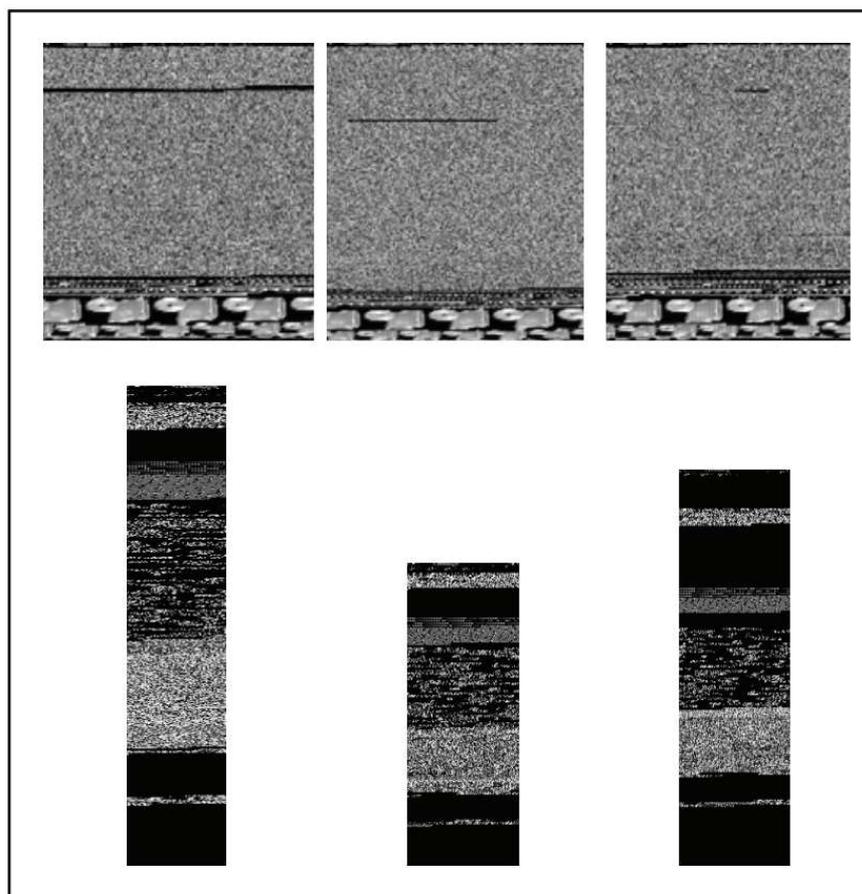


Figure 2.1: The images in the first row are images of 3 instances of malware belonging to the family Fakerean [4] and those in the second row belong to the family Dontovo.A [5].

This visual similarity is the result of reusing code to create new binaries. Thus, if old samples are re-used to implement new binaries, the resulting ones would be similar. In most cases, by representing an executable as a gray scale image it would be possible to detect small variations between samples belonging to the same family.

This visual similarity has been exploited by various authors for detecting and classifying malware. In particular, Nataraj *et al.* [31] extracted features from the gray scale representation of malware's binary content. Finally, a new executable is classified under one family or another using the K-Nearest Neighbor algorithm (K-NN) with the Euclidean distance as metric. Ahmadi

---

*et al.* [32] extracted Haralick and Local Binary Pattern features for classifying malware using boosting tree classifiers. Haralick features describe the correlation in intensity of pixels that are next to each other in space.

The grayscale image representation of software has some drawbacks directly related to how images are generated. Primarily, binaries are not 2-D images and by transforming them likewise you introduce unnecessary priors. First, to construct an image, an image width must be selected, which adds a new hyper-parameter to be tuned. By selecting the width consequently, one can determine the height on the image depending on the size of the binary. Second, it imposes non-existing spatial correlations between pixels in different rows, which might not be realistic.

Additionally, like the majority of static features, it suffers from code obfuscation techniques. In particular, techniques like encryption and compression might completely change the bytes structure of a binary program and, thus, methods based on this kind of representation would fail to correctly classify its class.

# Chapter 3

## Background

This section briefly describes the portable executable (PE) file format and its header. Moreover the necessary background is provided regarding Convolutional and Dense Neural Networks, which are used as the main deep learning algorithms for the purpose of this work.

### 3.1 Portable Executable Format

Each executable file has a common format called Common Object File Format (COFF), a format for executable, object code, shared library computer files used on Unix systems. Portable Executable format, Fig. 3.1, is one such COFF format available today for executable, object code, Dynamic-link libraries (DLLs), font files, and core dumps in 32-bit and 64-bit versions of Windows operating systems. It was introduced by Microsoft with the Windows NT 3.1 operating system. Unix uses the ELF format which is analogous to the Windows PE format. This thesis focuses on Windows PE files, explicitly.

The proposed model analyzes features extracted from PE files along with the image representation of the sample to determine whether the file is malicious or not. This section describes the information that can be obtained from PE files.

#### 3.1.1 MS-DOS Stub

This stub is executed whenever the file is executed in the MS-DOS environment. Its only purpose is to print a message indicating that the file cannot be run in the MS-DOS environment. A signature added after the MS-DOS stub indicates that the file is in PE format.

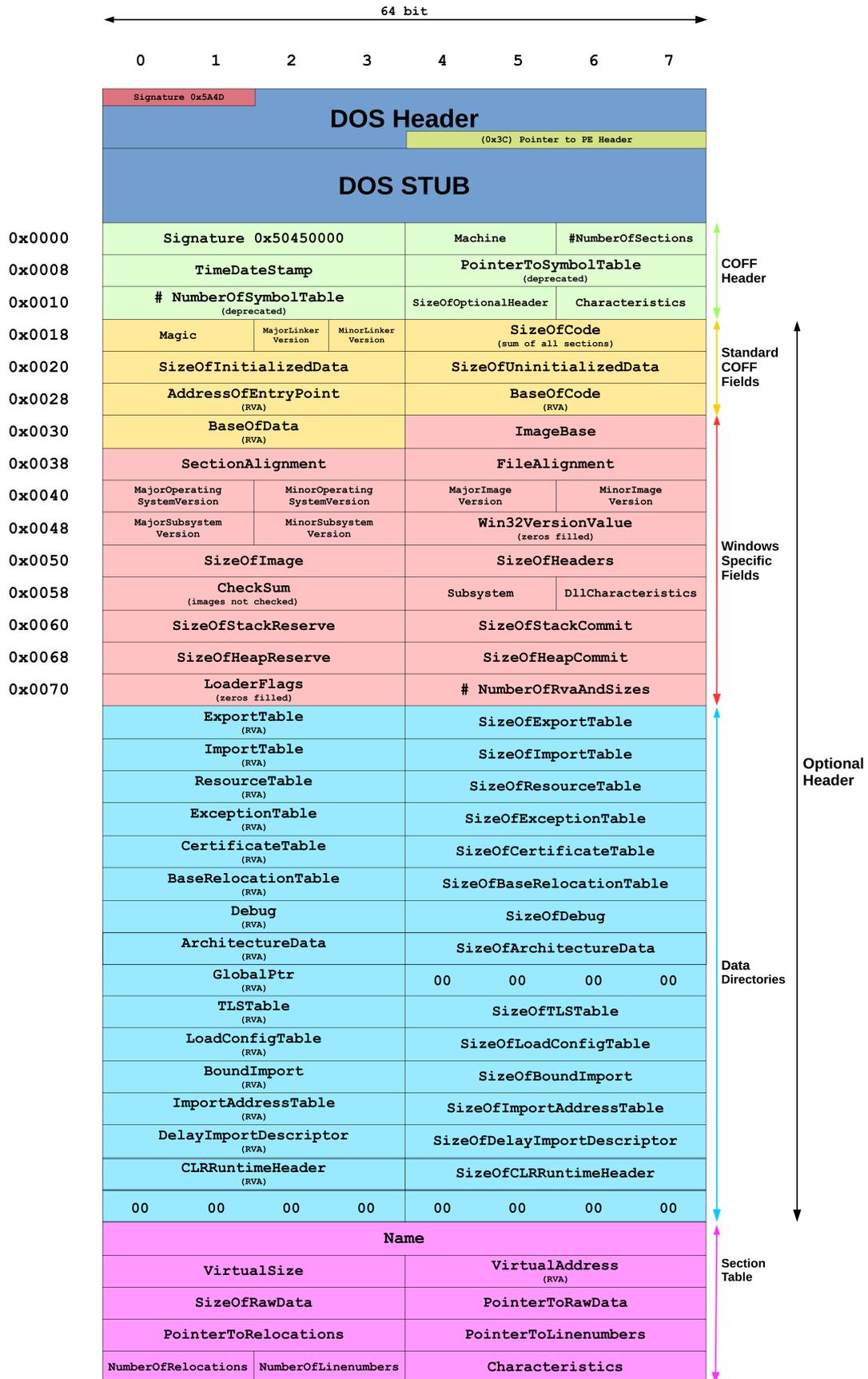


Figure 3.1: PE File Format [6].

### 3.1.2 COFF Header

At the beginning of an object file, or immediately after the signature of an image file, is a standard COFF file header. The format of this header is defined in Table 3.1. Note that the Windows loader limits the number of sections to 96.

A file can only be executed on a machine if the machine field matches the target machine the file is to be executed on.

Offset	Size	Field	Description
0	2	Machine	Identifies the target machine that the executable can run on.
2	2	NumberOfSections	Size of the section table.
4	4	TimeStamp	Date of Creation. Represented as seconds after January 1, 1970.
8	4	PointerToSymbolTable	File offset of COFF symbol table. 0 for no table.
12	4	NumberOfSymbols	Number of entries in the symbol table.
16	2	SizeOfOptionalHeader	Size of the optional header (required for executables)
18	2	Characteristics	Indicates the attributes of the file.

Table 3.1: COFF header.

### 3.1.3 Optional Header

The next 224 bytes in the executable file constitute the PE optional header. Despite its name, this is not an optional entry in PE executable files. This header provides information to the loader present in the operating system, which is responsible for handling the execution of files. The optional header is split into 3 major parts defined in Table 3.2.

Offset	Size	Header Part	Description
0	28	Standard fields	Common for Windows and Unix COFF implementations.
28	68	Windows-specific fields	Defines windows specific features.
96	Variable	Data directories	Address and size of special tables used by OS.

Table 3.2: Optional header parts.

Size of the optional header is defined in the *SizeOfOptionalHeader* field in the COFF header. A magic number present in the optional header determines whether the executable is PE32 or PE32+. PE32+ executables allow 64-bit memory address space, but can be no more than 2 gigabytes in size. Further details on the standard fields can be found in Table 3.3.

Offset	Size	Field	Description
0	2	Magic	0x10B for PE32 / 0x20B for PE32+.
2	1	MajorLinkerVersion	The linker major version number.
3	1	MinorLinkerVersion	The linker minor version number.
4	4	SizeOfCode	The size of the code (text) section.
8	4	SizeOfInitializedData	The size of the initialized data section.
12	4	SizeOfUninitializedData	The size of the uninitialized data section.
16	4	AddressOfEntryPoint	The memory address of the entry point relative to the image base.
20	4	BaseOfCode	The memory address that is relative to the beginning-of-code section.

Table 3.3: Optional header standard fields.

Windows specific fields contain certain information required specifically for Windows environments. It contains operating system version, image version, size of the headers, size of the image, DLL characteristics, loader flags, length of the data directory, the data directory itself, and the checksum. Size of the image determines how much address space must be reserved by the operating system for the image to run.

The data directories give the address and size of directories required by Windows. This includes, but is not limited to, import/export tables, resource table, exception table, etc.

### 3.1.4 Section Header

Section headers are located sequentially right after the optional header in the PE file format. Each section header is 40 bytes with no padding between them. Section headers are defined as in the following structure:

- **Name.** Each section header has a name field up to eight characters long, for which the first character must be a period.

- **PhysicalAddress or VirtualSize.** The second field is a union field that is not currently used.
- **VirtualAddress.** This field identifies the virtual address in the process address space where the section is loaded. The actual address is created by taking the value of this field and adding it to the ImageBase virtual address in the optional header structure.
- **SizeOfRawData.** This field indicates the FileAlignment-relative size of the section body. The actual size of the section body will be less than or equal to a multiple of FileAlignment in the file.
- **PointerToRawData.** This is an offset to the location of the section body in the file.
- **Characteristics.** Defines the section characteristics.

An application for Windows typically has the nine predefined sections named `.text`, `.bss`, `.rdata`, `.data`, `.rsrc`, `.edata`, `.idata`, `.pdata`, and `.debug`. Some applications do not need all of these sections, while others may define still more sections to suit their specific needs. Kindly refer to [33] for further information in predefined sections and the PE format.

## 3.2 Artificial Neural Networks

Artificial neural network is made up of neurons, which are connected with each other and form a neural network. A neuron is the unit of a network and the basic structure of a single neuron is shown in Fig. 3.2.

According to the figure given, suppose the neuron takes  $x_i$  as the input and gets the outputs based on the following computation:

$$output = f \left( \sum_{i=1}^3 (w_i x_i) + b \right), \quad (3.1)$$

where  $w_i$  are defined as weights, the  $b$  is defined as bias and  $f(.)$  is a non-linear activation function. During the computation phase, every input value

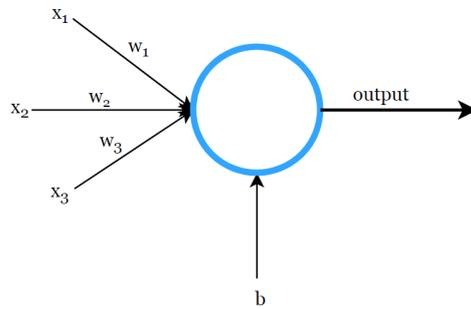


Figure 3.2: Basic structure of a neuron.

$x_i$  is weighted and multiplied by a weight  $w_i$  then the weighted input values plus the bias  $b$  are devoted into the activation function, where this linear combination is transformed into a non-linear one. There exist several classic non-linear activation functions, among which logistic sigmoid function and hyperbolic tangent function, presented in Fig. 3.3 are general choices.

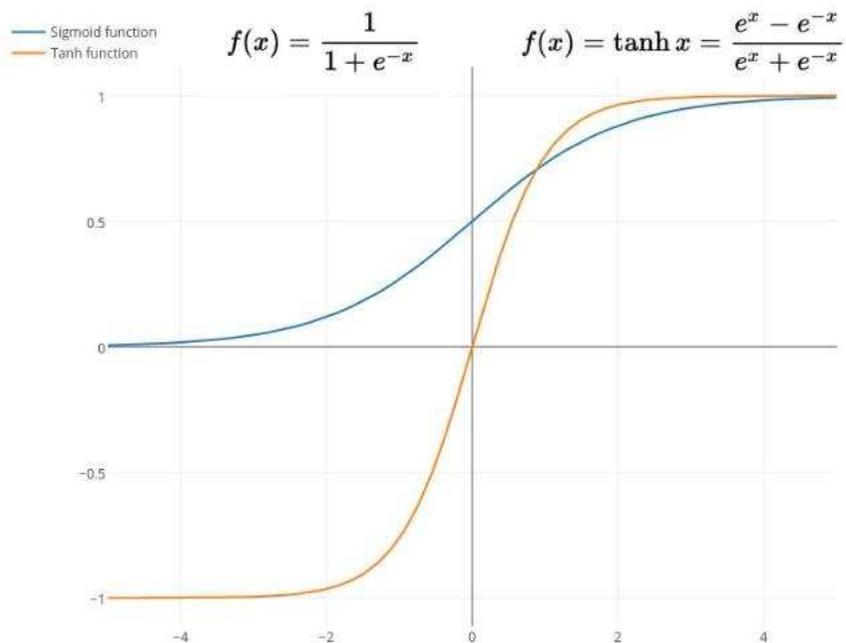


Figure 3.3: Sigmoid function and Hyperbolic tangent function.

According to the function definition, for one single neuron, the mapping

relations between input and output is in fact a logistic regression. A classic neural network is made up of multiple neurons and the outputs of the previous layer are the inputs of the next layer. Fig. 3.4 is an example of a feedforward neural network, which is also known as multi-layer perceptron. As it is shown,

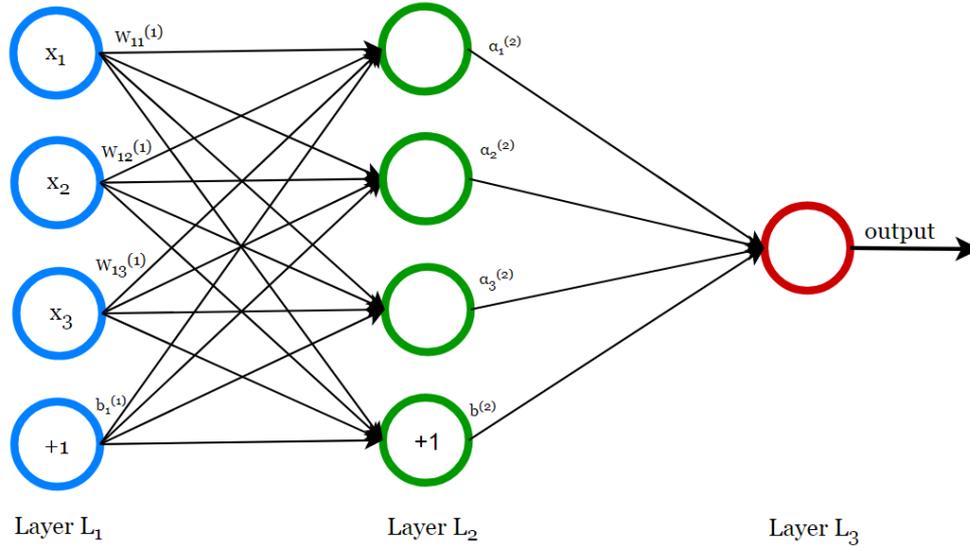


Figure 3.4: Basic structure of multi-layer perceptron.

within the neural network, the neurons are grouped into layers. Each layer is fully connected to the subsequent one and the connections do not form cycles. It is clear to see that within each layer, there is a bias parameter  $b_i$ , which is used to compute the output of the corresponding neuron. The leftmost layer is the input layer and the rightmost layer is the output layer. The layer in the middle is defined as hidden layer, as its values can not be observed in the training set.

Suppose there are three layers within the neural network,  $n_1=3$ , and the  $i$ -th layer is labeled as  $L_i$ . Therefore, the first layer, which is also defined as the input layer, is presented as  $L_1$ , the second layer as  $L_2$  and the third layer as  $L_3$ . The parameters of the model are defined as  $(W, b) = (W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)})$ , where  $b_i^{(l)}$ , is the bias to unit  $i$  in layer  $l+1$  and  $W_{ij}^{(l)}$  denote the weight value to the connection between the unit  $j$  in the layer  $l$  and the unit  $i$  in layer  $l+1$ . Moreover,  $W^{(1)} \in R^{3 \times 3}$ ,  $W^{(2)} \in R^{1 \times 3}$ ,  $b^{(1)} \in R^{3 \times 1}$  and  $b^{(2)} \in R^1$  are defined in this case.

In the initial step, the activations of the hidden nodes are computed as:

$$\alpha_1^{(2)} = f(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + b_1^{(1)}), \quad (3.2)$$

$$\alpha_2^{(2)} = f(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + b_2^{(1)}), \quad (3.3)$$

$$\alpha_3^{(2)} = f(W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3 + b_3^{(1)}), \quad (3.4)$$

where  $\alpha_i^{(l)}$  is defined as the activation of unit  $i$  in layer  $l$ , and  $f(\cdot)$  is a non-linear activation function. The activations of the hidden units are then devoted into the next layer, which takes all these activations as the inputs. Therefore, the output of the last layer is computed as:

$$output = \alpha^{(3)} = \sigma(W^{(2)}\alpha^{(2)} + b^{(2)}) = \sigma(W^{(2)}f(W^{(1)}x + b^{(1)}) + b^{(2)}), \quad (3.5)$$

where  $\sigma$  means that the activation function for the output layer could be different than the activation function in the hidden units.

The whole process is called forward propagation, based on the fact that the inputs are forwarded through the network. There is one thing to stress that the non-linear activation function is a must within the network, which transforms the original linear combination into a non-linear one.

In a supervised learning scenario, Artificial Neural Networks can be trained using the backpropagation algorithm, which readjusts the weights of the interconnections in the neural network based on local error rates.

Backpropagation in neural networks describes the process of using a local error of the network to readjust the weights of the interconnections backwards through the neural net. Explicitly, this means that after a prediction for a set of input values has been made, the actual output value is compared to the prediction value and an error is calculated. This error is then used to readjust the weights of the connections starting at the edges that are directly connected to the output nodes of the network and then proceeding further into it. In order to train a neural network, it is important to understand the main parameters that can be used to optimize the learning process:

The **learning rate** specifies how fast the learning process is performed. The parameter's value lies between 0 and 1 and is multiplied with the local

error for every output value. Therefore, a learning rate of 0 would result in no adaptation at all. The correct setting for the learning rate is crucial to the success of the learning process. If the value is set too high the weights can oscillate and complicate the finding of the optimal values. However, if the value is set too low, found errors will not have enough weight to push the network into a new optimization and the weights can get stuck in local maximum. In order to find the correct settings, a decay parameter can be added. This parameter ensures a high learning rate value in the earlier cycles of the training process to avoid getting stuck in local maximum and forces its reduction during the learning process to avoid oscillation.

Another important parameter for neural networks is called **momentum**. It is used to smooth out the optimization process by using a fraction of the last weight change and adding it to the new weight change.

The **minimal error** is a stop criterion for the learning process. Once the combined error of the network falls below this threshold, the learning process is stopped.

Combining these parameters, the formula for computing a new weight for a connection is the following:

$$W = l + \epsilon + m * W_p, \quad (3.6)$$

where  $W$  is the new weight change,  $l$  is the learning rate,  $\epsilon$  is the minimal error,  $m$  is the momentum and  $W_p$  is the weight change of the previous cycle.

### 3.3 Convolutional Neural Networks

Convolutional Neural Networks, known as CNNs, are a category of Neural Networks that have proven very effective in areas such as image recognition and classification. CNNs have been successful in identifying faces, objects and traffic signs apart from powering vision in robots and self driving cars. There are four main layers in the CNN structure: convolution, non-linear, pooling or subsampling and fully connected layer. The mentioned layers are depicted in Fig. 3.5.

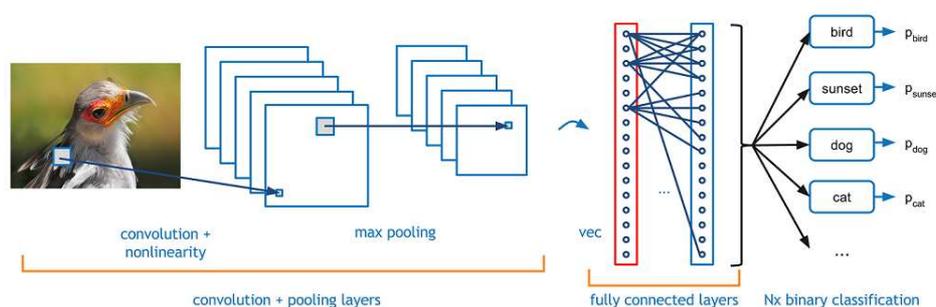


Figure 3.5: Example of a convolutional neural network.

### 3.3.1 Convolution Layer

The convolution layer is the foremost part of CNN, which does the heavy calculation of convolution neural network operation. The CNN extracts different features of the inputs and the convolution layer extracts low-level features of the image, such as edges, lines, and corners. The key point of the convolution layers is the usage of learnable kernels. The kernels are generally small in spatial dimensionality, but they can spread along the entire input. When an input comes into a convolutional layer, the layer applies each filter across the input and then produces a 2-dimensional activation map. Every kernel has its own activation map, which will be stacked along the depth later. As a result, it is necessary to stress that the depth of the filter should be the same as the depth of the input.

Fig. 3.6 visualizes the classic operation of the kernel within the convolution layer. After extraction of the target part from the input, the kernel passes through the entire vector. After gliding through the input, the output of convolution calculation is the scalar product of each value in the kernel.

Generally, the kernel starts from the top left corner of the image. Hypothetically, there is an input with  $32 \times 32$  array of pixel values and the kernel covers the area of  $5 \times 5$ . As the filter is sliding around the input image, it multiplies the values in the kernel with the original pixel values of the image and then the multiplication is summed up and every part of the input volume produces its own number. After the kernel passing through all the parts of the image, the output of the convolution operation is an array with

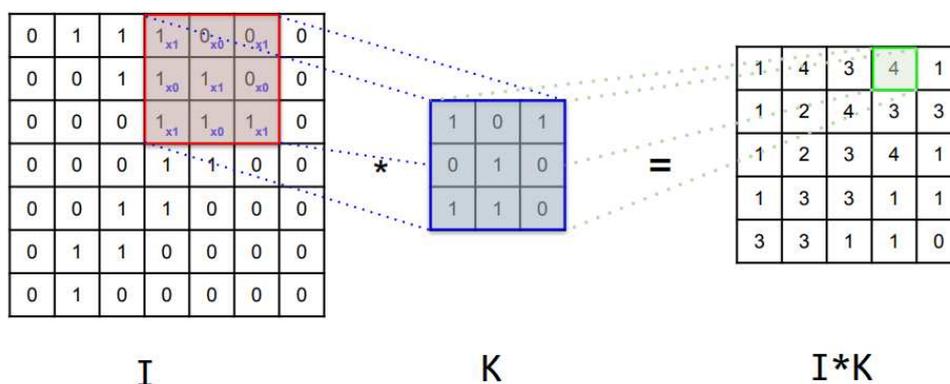


Figure 3.6: Example of kernel calculation within convolution layer.

a  $28 \times 28$ . This array is known as the feature map. Compared with artificial neural network, the convolutional layer shows great ability in reducing the complexity of the model by using the kernel.

When designing the convolution layer, there are three hyper-parameters to be considered: the depth, the stride and setting zero-padding.

The depth of the output volume corresponds to the number of the kernels used for convolution and each of these kernels offers its own feature map of the image input. By reducing this hyper parameter, it can minimize the total number of neurons within the network. However, it may decrease the performance of the convolution neural network on the pattern recognition.

The stride illustrates the steps of kernel sliding through the input volume. If the stride is set as 1, then the filters will move one pixel at a time. If the stride is set as 2, then the filters will jump 2 pixels at a time during the sliding period, which will produce smaller output volumes. It is clear to see that if the stride is set to a greater number, it will reduce the amount of overlapping area and produce lower spatial dimension outputs.

In general cases, it is convenient to fill zeros around the border of the input, which is called zero-padding. The introduction of zero-padding gives further control to the spatial size of the output volumes.

When using these parameters, the spatial dimensionality of the convolutional layers output is changed. The formula followed gives the details of this

change:

$$\frac{(V - R) + 2Z}{S + 1}, \quad (3.7)$$

where  $V$  represents the input volume size,  $R$  represents the receptive field size,  $Z$  is the amount of zero padding set and  $S$  represents the stride.

It is necessary to notice that if the calculation from this equation is not equal to an integer, then the stride needs to be altered to meet this expectation, or the neuron is not able to fit for the given input.

Despite the effort of these methods, in some cases, the model is still enormous as some images may have multiple dimension. For this consideration, parameter sharing is used to reduce the overall number of parameters within the convolutional layer. The idea of parameter sharing is based on an assumption that if the feature of one region is useful to compute at a set spatial region, then it will be likely useful in another region. Within the convolution layer, each activation map within the output volume is set as the same weights and bias, so there is a huge decrease on the number of the parameters produced by the convolutional layer. Based on this theory, during the phase of backpropagation, each neuron in the output represents the overall gradient so that only a small group of weights need to be updated instead of every single one.

### 3.3.2 Non-linear Layer

The convolutional neural network applies a non-linear transformation on the input, whose purpose is to identify the features within each hidden layer. In artificial neural network, the non-linear transformation function is sigmoid or hyperbolic tangent. However, for image processing, if data are more sparse, the result will be better. Based on this understanding, rectified linear units is often used as the non-linear transformation.

Rectified linear unit, implements function:  $y = \max(x, 0)$ , so the output is in the same size as the input. Rectified linear unit increases the non-linear properties of the decision function and it has no negative effect on the receptive fields of the convolution layer. Compared to other non-linear functions, the training speed of the rectified linear unit is much faster. Fig.

3.7 sets an example of the rectified linear units.

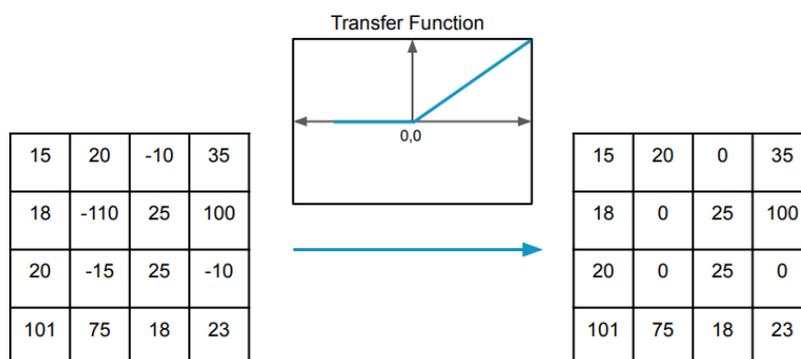


Figure 3.7: Example of rectified linear unit transformation.

### 3.3.3 Pooling Layer

After the operation of convolution layer, the data comes into the pooling layer. The major purpose of the pooling layer is also to reduce the dimensionality, the number of the parameters as well as the computational complexity. Besides, it helps to make the features robust against noise and distortion. The pooling layer operates on each feature map of the input and scales its dimensionality by using the function defined. Generally, there are two classic pooling functions, which are max pooling and average pooling. Fig. 3.8 illustrates the operations of both pooling methods.

The pooling layer applies the max pooling function with a  $2 \times 2$  kernel and a stride of 2 along the spatial dimensions of the input. It is based on the concern of the destructive functionality of pooling layer. By this operation, it reduces the feature map down to 25% of the original size, while it still maintains the depth volume to its standard size. In addition, it allows the layer to pass through the entire spatial dimensionality of the input with the overlapping area to be utilized. If the stride is set to 3 with a kernel size set to 3, it will effectively decrease the performance of the model.

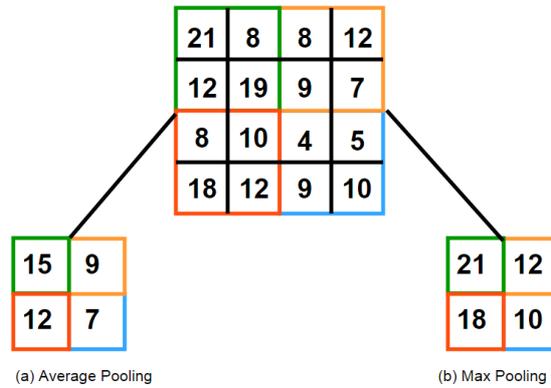


Figure 3.8: Two classic methods for pooling.

### 3.3.4 Fully Connected Layer

After several repeats of the previous layers, the data comes to the final layer of the convolution neural network, which is the fully connected layer. Within the fully connected layer, the neurons are directly connected to the neurons in the two adjacent layers. The aim of this layer is to sum up the weights of the features coming from the previous layers and indicates the probability of each class. For example, if there is a convolution neural network for gender classification, and the output vector is a probability of  $[0.7, 0.3]$ , it means there is 70% probability of male gender and 30% for female gender.

The functionality of each layer within the convolution neural network has been explained. A classic convolution neural network basically contains two parts. One part is several repeats of convolution layer, non-linear layer, pooling layer. The purpose of this part is to reduce the dimensionality of the input volume. Another part is the fully connected layer, following the previous repeated layers, used to convert any vector of real numbers into a vector of probabilities.

# Chapter 4

## Implementation

In this chapter, the implementation methodology is analyzed. Initially, data collection techniques will be discussed and the lab environment is presented. Next, the dataset is described along with the conversion of samples to images. Finally, the various classifiers are introduced.

The development is based on Python, due to its simplicity and flexibility of use in machine learning applications.

### 4.1 Data Collection

Malicious samples were collected from the MalShare Project [1], which is a collaborative effort to create a community-driven public malware repository. An API is provided for registered users to allow access to files and data. For this thesis, a command-line tool was developed to interact with the API of the platform to make the data collection process efficient and automated. More information about the tool can be found in the Github repository of this thesis [3].

Unfortunately, not all files in MalShare are considered as malware. Therefore, another command-line tool was developed to check the maliciousness of a sample. This tool takes advantage of the VirusTotal API. VirusTotal aggregates many antivirus products and online scan engines to check for viruses that the user's antivirus may have missed, or to verify against any false positives. For this work, malware is considered a file which is flagged as malicious by 5 antivirus products or more in VirusTotal. Further information regarding the tool can be found in the Github repository of this thesis [3].

Benign samples were collected from a personal computer. They consisted of default Windows executables, DLLs and clean installations of popular

software like Skype, Mozilla Firefox, Dropbox, VLC Player and more. Checks for maliciousness were performed on those samples as well, for clarity. In Table 4.1, the exact number of samples is presented.

Total Samples	21,438
Malicious	12,863
Benign	8,575

Table 4.1: Samples collected.

## 4.2 Lab Environment

Before trying to analyze any kind of malware, a proper environment set up is needed. This is the most efficient way to collect information from the malicious executable without getting a computer infected. The best thing to do in such cases is to have a virtual machine image ready for testing purposes. System settings of the virtual machine are shown in Table 4.2.

Operation System	Windows 10
Base Memory	10GB
Storage	200GB
Processor	Intel Core i7 (2.93GHz) 3/8 cores

Table 4.2: Virtual Machine System Settings.

The virtual machine was used to store all the samples in an isolated environment, run test scripts, extract the features of the dataset and convert the samples to grayscale images.

## 4.3 Extracted Features and Dataset

There are two main components of the PE file extracted in order to train our model - parsed information from a) the headers of its sample and b) the

encoded raw byte information.

A convenient Python script was developed, using the LIEF [34] and numpy [35] modules, to extract the required data from any given PE file. In this module, the extracted features are described. In total, 486 features were extracted.

The first 11 scalars of the vector encode a set of boolean properties that LIEF parses from the PE, as presented in Table 4.3. Each property will be encoded to a 1.0 if true, or to a 0.0 if false.

Property	Description
has_configuration	True if the PE has a Load Configuration.
has_debug	True if the PE has a Debug section.
has_exceptions	True if the PE is using exceptions.
has_exports	True if the PE has any exported symbol.
has_imprrts	True if the PE is importing any symbol.
has_nx	True if the PE has the NX bit set.
has_relocations	True if the PE has relocation entries.
has_resources	True if the PE has any resource.
has_rich_header	True if there is build information.
has_signature	True if the PE is digitally signed.
has_tls	True if the PE is using TLS

Table 4.3: Important properties of a file.

As presented in Fig. 4.1, properties like debug, exports and signature are important features regarding malware detection, while the exceptions property could be omitted during the training phase.

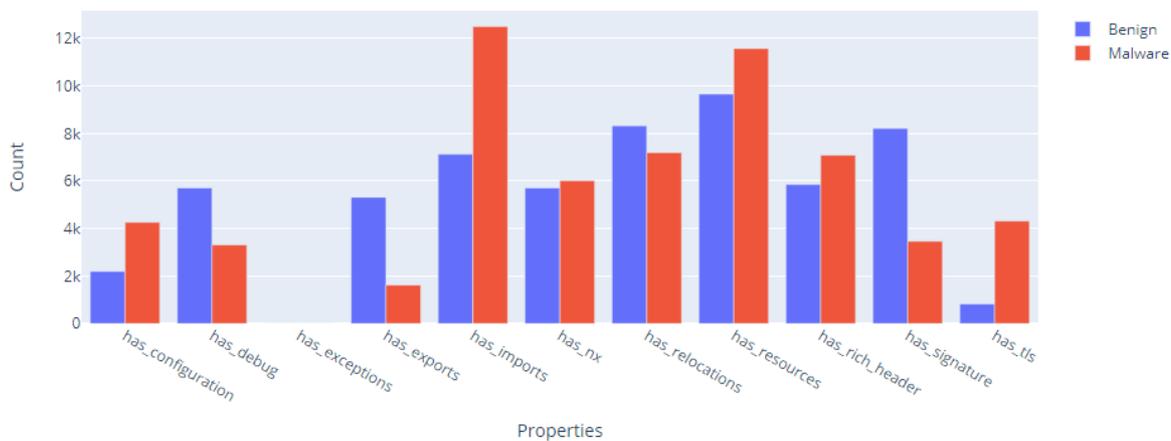


Figure 4.1: Different properties along samples.

Then, 64 elements follow, representing the first 64 bytes of the PE entry point function, each normalized to  $[0.0,1.0]$  by dividing each of them by 255 - this will help the model detecting those executables which have very distinctive entrypoints that only vary slightly among different samples of the same family. The conclusion from Fig. 4.2 is that in most byte positions, malware samples tend to have higher values.

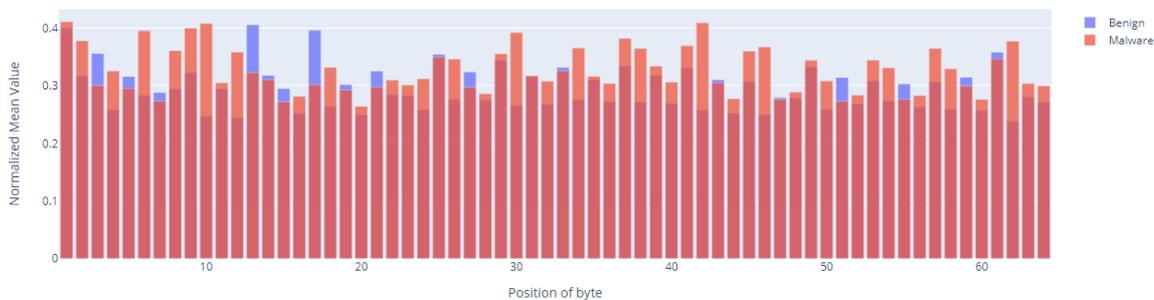


Figure 4.2: Distribution of byte values of the the entry point.

The next 256 values refer to a histogram of the repetitions of each byte of the ASCII table in the binary file - this data point will encode basic statistical information about the raw contents of the file.

The API being used by the PE is quite relevant information. Therefore, 150 most common libraries were selected manually to be included in the dataset as features. For each API being used by the PE, the column of

the relative library is incremented by one, creating another histogram of 150 values then normalized by the total amount of API being imported.

The ratio of the PE size on disk and the size it'll have in memory, its virtual size, is also encoded. In Fig. 4.3 the difference between malicious and benign samples is depicted.

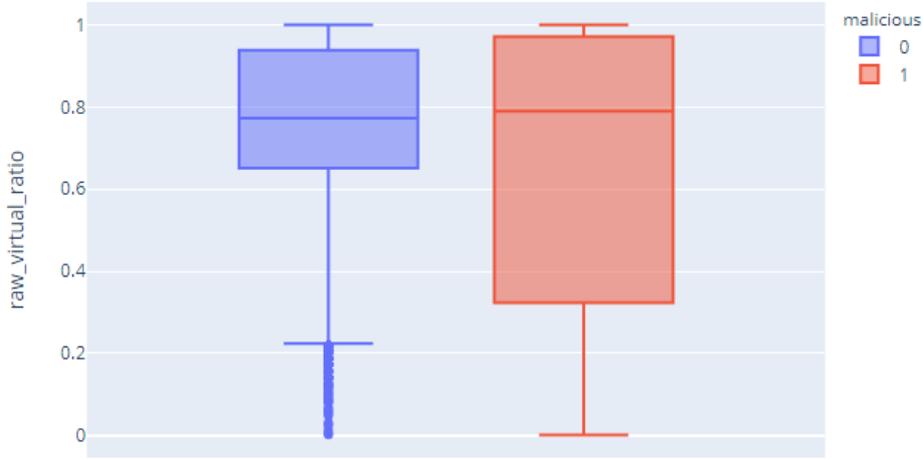


Figure 4.3: Ratio of virtual size and disk size.

The last features include information about the PE sections, normalized to  $[0.0, 1.0]$ ; namely the amount of sections containing code and the ones containing data, the sections marked as executable, the average Shannon entropy of each one and the average ratio of their virtual size and disk size. These datapoints will tell the model if and how the PE is packed/compressed/obfuscated as mentioned in paragraph 2.1.2.

To compute the average Shannon's Entropy across all sections, the following operation was performed:

$$\frac{\sum_n H(n)}{H_{max}},$$

where  $n$  is defined as the number of sections,  $H(n)$  represents Shannon's entropy of section  $n$  and  $H_{max}$  is the max entropy across all sections. A full overview of the dataset's structure is presented in Table 4.4.

Offset	Description
0	md5 checksum
[1:11]	Important boolean properties.
[12:77]	First 64 bytes of entry point, normalized to [0.0,1.0].
[78:332]	Encoded histogram of each byte frequency, normalized to [0.0,1.0].
[333:482]	Encoded library calls.
483	Ratio between Virtual Size and Raw Size of the executable.
484	Percentage regarding the sections of the PE referring to code execution.
485	Percentage regarding the sections of the PE referring to memory execution.
486	Average entropy of the file.
487	Ratio between section size and virtual size
488	1 if malicious, else 0.

Table 4.4: Dataset structure.

## 4.4 Samples to Images

Once the dataset was gathered, the samples were converted to a 120x120 grayscale image, which means that each pixel has a value between 0 and 255. The steps are the following:

- Read the sample in bytes.
- Remove extra bytes in case the size of the sample is not multiple of 120.
- Convert each byte to its corresponding integer.
- Reshape the image to 120x120, using linear interpolation.

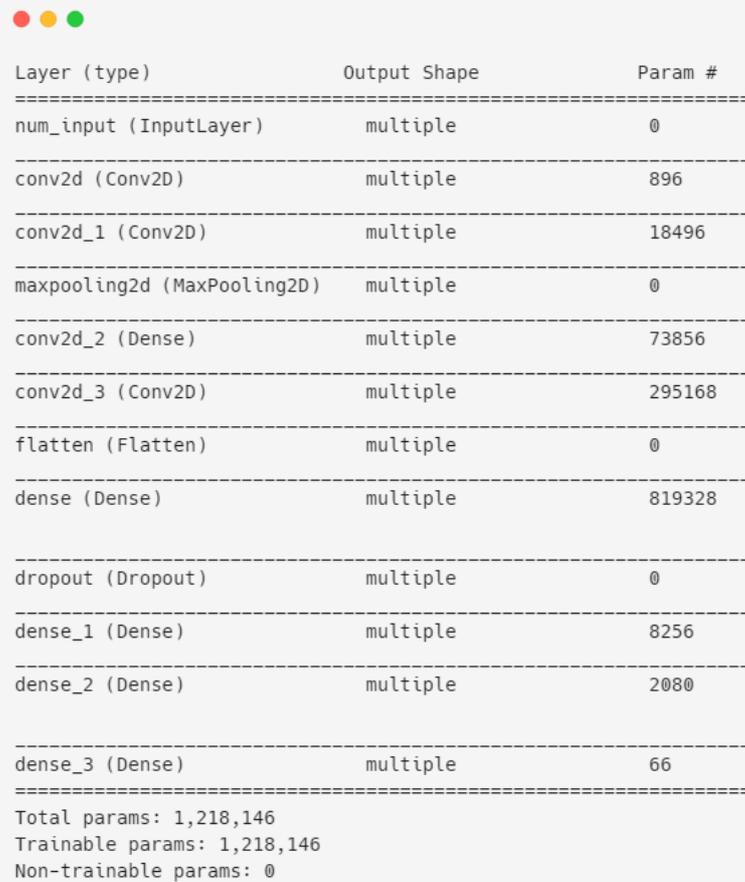
In this thesis, we will investigate the effectiveness of combining the image representation of the samples with the features extracted from the PE file.

---

## 4.5 Classifiers

Deep neural networks were used for analyzing the data. There were three neural networks constructed, one based on CNNs, one based on dense layers and a third one combining convolutional and dense layers together. The rectified linear unit activation function has been used for these networks. The Adam [36] optimizer implemented in the Keras [37] library was used for gradient-based optimization of our classifiers. A summary of both neural networks are shown in Figs. 4.4, 4.5, 4.6, respectively.

Each neural network has a different input. The image representation of a sample is passed as input in the convolutional neural network to predict its maliciousness. The dense neural network takes advantage of the encoded features described in 4.3 to identify a file as malware. Finally, the proposed enhanced neural network is testing the effectiveness of using both an image of the sample and numerical features as input to classify its target.

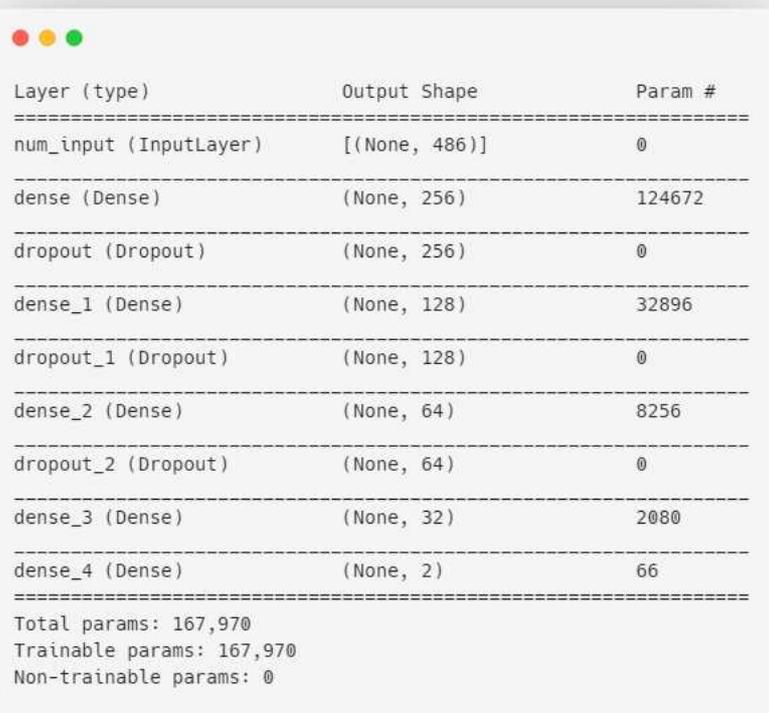


```
Layer (type)                Output Shape                Param #
-----
num_input (InputLayer)      multiple                    0
conv2d (Conv2D)             multiple                    896
conv2d_1 (Conv2D)           multiple                    18496
maxpooling2d (MaxPooling2D) multiple                    0
conv2d_2 (Dense)            multiple                    73856
conv2d_3 (Conv2D)           multiple                    295168
flatten (Flatten)           multiple                    0
dense (Dense)                multiple                    819328

dropout (Dropout)           multiple                    0
dense_1 (Dense)              multiple                    8256
dense_2 (Dense)              multiple                    2080

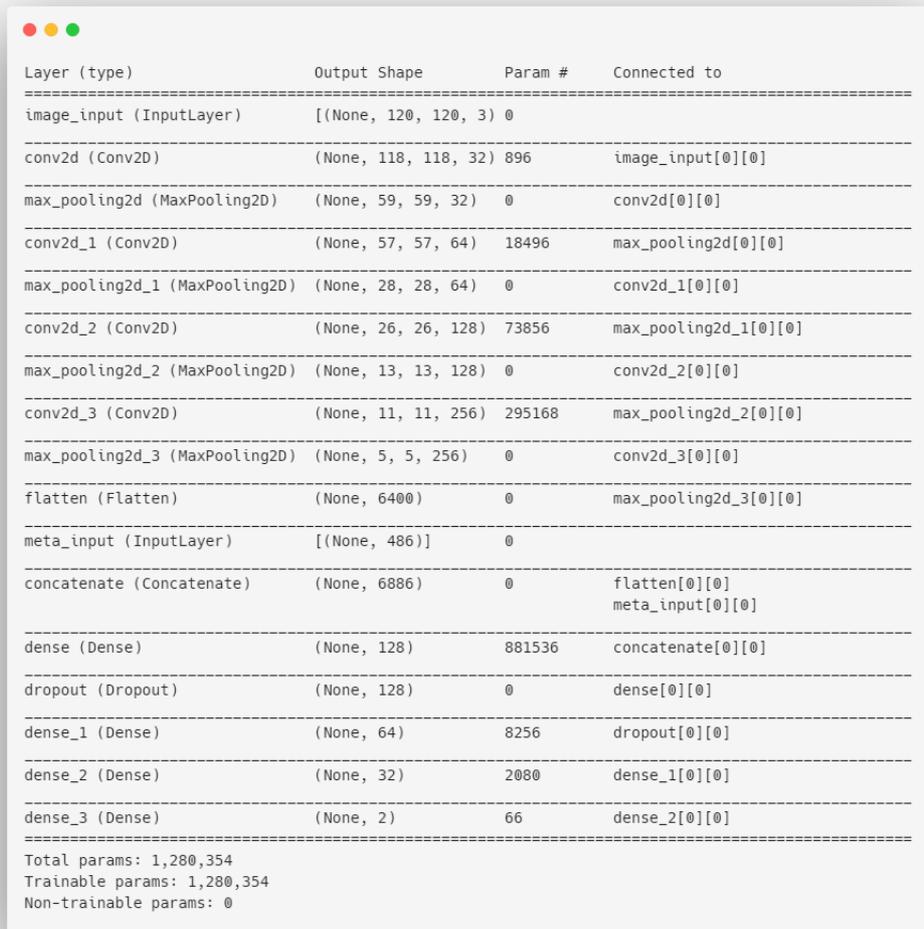
dense_3 (Dense)              multiple                    66
-----
Total params: 1,218,146
Trainable params: 1,218,146
Non-trainable params: 0
```

Figure 4.4: Summary of the Convolutional Neural Network.



```
Layer (type)                Output Shape                Param #
=====
num_input (InputLayer)      [(None, 486)]              0
-----
dense (Dense)                (None, 256)                124672
-----
dropout (Dropout)           (None, 256)                0
-----
dense_1 (Dense)              (None, 128)                32896
-----
dropout_1 (Dropout)         (None, 128)                0
-----
dense_2 (Dense)              (None, 64)                 8256
-----
dropout_2 (Dropout)         (None, 64)                 0
-----
dense_3 (Dense)              (None, 32)                 2080
-----
dense_4 (Dense)              (None, 2)                  66
=====
Total params: 167,970
Trainable params: 167,970
Non-trainable params: 0
```

Figure 4.5: Summary of the Dense Neural Network.



```

Layer (type)                Output Shape                Param #   Connected to
-----
image_input (InputLayer)    [(None, 120, 120, 3)] 0
conv2d (Conv2D)             (None, 118, 118, 32) 896      image_input[0][0]
max_pooling2d (MaxPooling2D) (None, 59, 59, 32) 0        conv2d[0][0]
conv2d_1 (Conv2D)           (None, 57, 57, 64) 18496    max_pooling2d[0][0]
max_pooling2d_1 (MaxPooling2D) (None, 28, 28, 64) 0        conv2d_1[0][0]
conv2d_2 (Conv2D)           (None, 26, 26, 128) 73856    max_pooling2d_1[0][0]
max_pooling2d_2 (MaxPooling2D) (None, 13, 13, 128) 0        conv2d_2[0][0]
conv2d_3 (Conv2D)           (None, 11, 11, 256) 295168   max_pooling2d_2[0][0]
max_pooling2d_3 (MaxPooling2D) (None, 5, 5, 256) 0        conv2d_3[0][0]
flatten (Flatten)           (None, 6400) 0        max_pooling2d_3[0][0]
meta_input (InputLayer)    [(None, 486)] 0
concatenate (Concatenate)   (None, 6886) 0        flatten[0][0]
                                meta_input[0][0]
dense (Dense)               (None, 128) 881536   concatenate[0][0]
dropout (Dropout)          (None, 128) 0        dense[0][0]
dense_1 (Dense)            (None, 64) 8256     dropout[0][0]
dense_2 (Dense)            (None, 32) 2080     dense_1[0][0]
dense_3 (Dense)            (None, 2) 66       dense_2[0][0]
=====
Total params: 1,280,354
Trainable params: 1,280,354
Non-trainable params: 0

```

Figure 4.6: Summary of the Enhanced Neural Network.

# Chapter 5

## Experiments and Results

In this chapter, the experiments are presented. Results are discussed and the performance of the classifiers is compared.

### 5.1 Experimental Setup

Our classifiers were trained on a computer with an Intel i7 K875 processor and 20GB of RAM. Implementation was in Python with the following libraries installed:

- TensorFlow.
- NumPy.
- Pandas.
- scikit-learn.
- LIEF.
- PILLOW.

There are also packages and libraries which the above libraries depend on, but are typically installed automatically as part of the installation process. Anaconda Python was used with Python 3.7.6 for all experiments.

### 5.2 Metrics

In this section, the metrics to be used for this purpose are identified. The aim of this work is to test the accuracy and diagnostic ability of our model. For

this purpose, the receiver operating characteristic (ROC) curve is plotted and the area under curve (AUC) is computed. This method generally provides a better measure of the diagnostic ability of a classifier as compared to simply stating the overall accuracy of the model against a given test set.

The ROC curve is derived by plotting the true positive rate (TPR) of a classifier against the false positive rate (FPR). The TPR and FPR are calculated using the following equations:

$$TPR = \frac{TP}{P} = \frac{TP}{TP + FN},$$

$$FPR = \frac{FP}{P} = \frac{FP}{FP + TN},$$

where  $TP$  is true positives,  $P$  is the total positive samples present in the test set, and  $FN$  is false negatives.  $FP$  is false positives, and  $TN$  is true negatives.

## 5.3 Results

In this section, validation and test results are presented for each classifier. The data was split into training and test sets. Due to the small amount of data available, a 90-10 split was chosen. The epoch number is set to 140, since this was the max number of epochs the classifiers needed to reach convergence. Overfitting events were captured and presented in the figures of this section. Multiple training sessions took place to test the validity of the results. In the subsections following, accuracy and loss plots are provided for each classifier.

Loss is defined as the difference between the predicted value by your model and the true value. The most common loss function used in deep neural networks has been used, cross-entropy. It is defined as:

$$Cross - entropy = - \sum_{i=1}^n \sum_{j=1}^m y_{i,j} \log(p_{i,j}), \quad (5.1)$$

where  $y_{i,j}$  denotes the true value i.e. 1 for malware and 0 otherwise, and

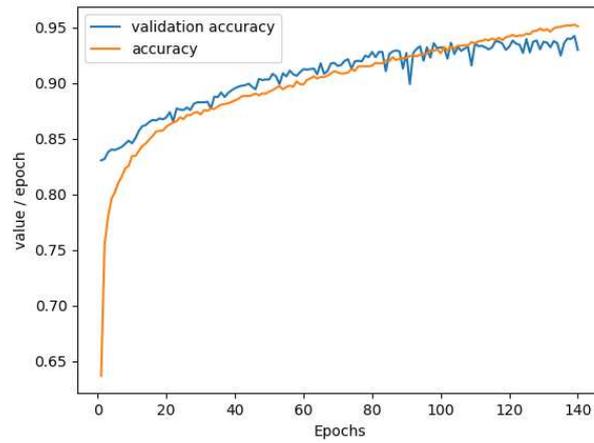
---

$p_{i,j}$  denotes the probability predicted by your model of sample  $i$  belonging to class  $j$ . Accuracy measures the performance of our models. It is defined as:

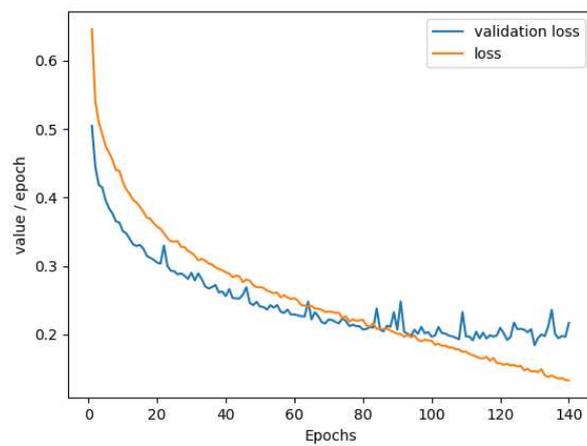
$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \quad (5.2)$$

### 5.3.1 CNN Classifier

The results from the training of CNN classifier can be found in Figs. 5.1a, 5.1b. The model reaches convergence near epoch 94. The spikes are an unavoidable consequence of the Adam optimizer and its function. During every epoch, a different batch of samples is validated. However, batch size is not equal to the cardinality of your training set. Thus, fluctuations are observed in the validation loss. To double check the validity of the results, the mean validation loss was calculated equal to 0.25. Based on 5.1b this value belongs to the curve. Therefore, the results are not affected by these spikes.



(a) Accuracy

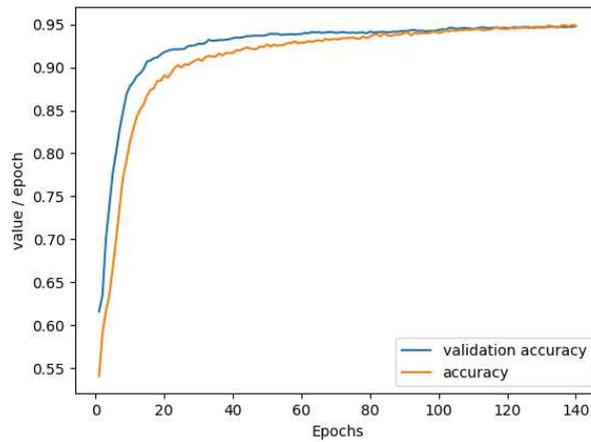


(b) Loss

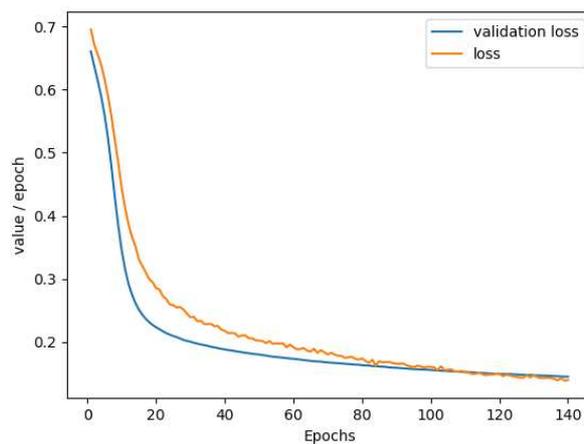
Figure 5.1: CNN Results.

### 5.3.2 DNN Classifier

The results from the training of DNN classifier are presented in Figs. 5.2a, 5.2b. This model reaches convergence at a later epoch, around 120, compared to the CNN, due to less parametric complexity. However, the encoded features have enabled the model to reach higher accuracy and lower loss as they are more definite than the image based features.



(a) Accuracy



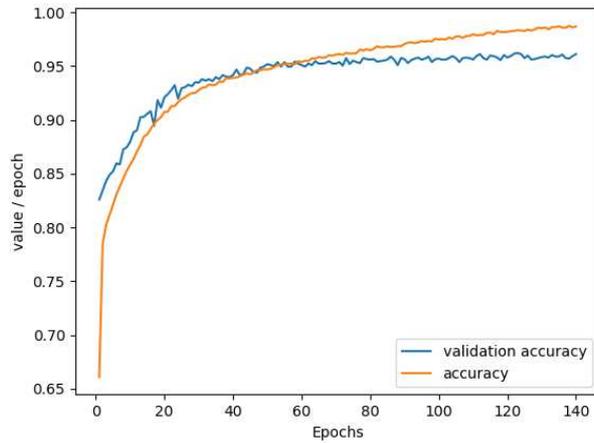
(b) Loss

Figure 5.2: DNN Results.

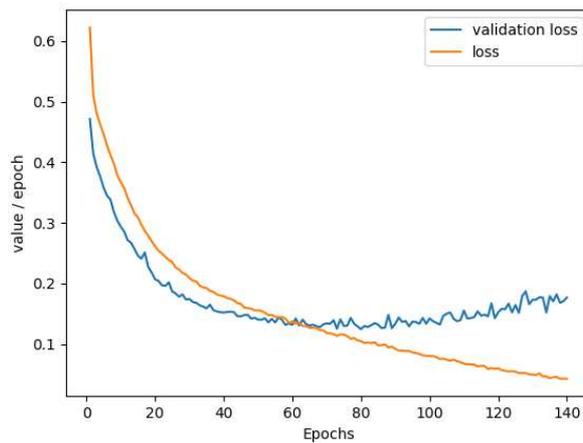
### 5.3.3 Enhanced Classifier

It is observed based on Figs 5.3a, 5.2a, 5.1a that the current classifier starts with almost the same validation accuracy with the CNN classifier and reaches a higher maximum validation accuracy than the DNN classifier. The unexpected notice is that in this case, the classifier reached convergence earlier than the others. Therefore, in later epochs, overfitting events occurred.

Also, validation loss has the minimum starting value of all classifiers and reaches lower values as epochs go by.



(a) Accuracy



(b) Loss

Figure 5.3: Enhanced Classifier Results.

### 5.3.4 Malware in the wild

In order to validate the efficiency of the enhanced classifier, another experiment was conducted. With the tools described in 4.1, another 1,000 samples were collected. This dataset was totally unknown to the classifier. The purpose of the experiment was to check the TPR between the most popular antivirus products and our neural network. The results are presented in Table 5.1.

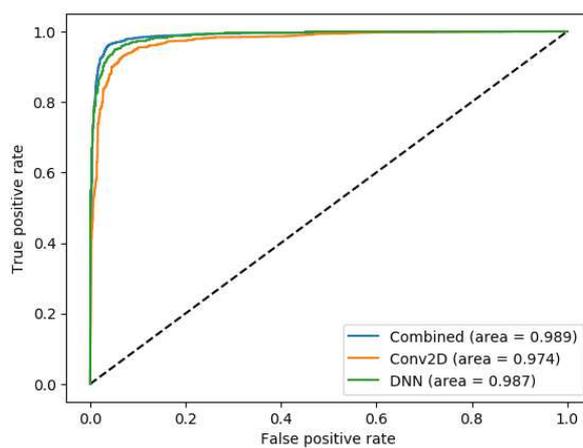
Product	TP	TN	FN	TPR
Avast	896	8	95	0.904
AVG	930	8	61	0.938
BitDefender	946	8	45	0.955
ESET-NOD32	973	8	18	0.982
FireEye	962	8	29	0.971
Kaspersky	945	8	46	0.954
McAfee	958	8	33	0.967
Microsoft	963	8	28	0.972
Symantec	941	8	50	0.95
TrendMicro	876	8	115	0.884
<b>This Thesis</b>	962	1	29	0.971

Table 5.1: Results in unknown samples.

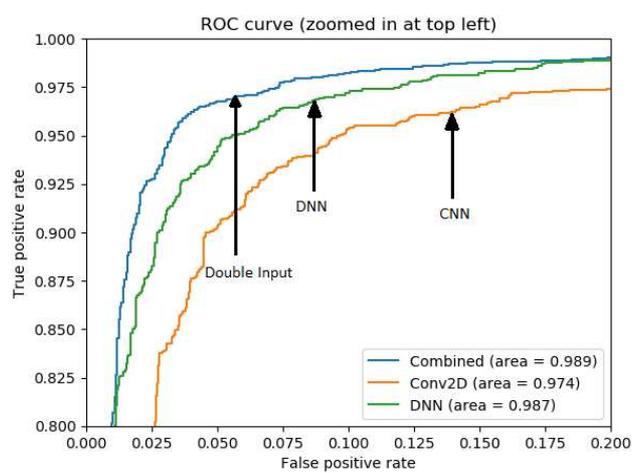
It is observed that the neural network scored the third highest TPR. Further training with more data would improve these results.

### 5.3.5 Overall Results

Comparing the three classifiers in Figs. 5.4a, 5.4b, it is proven that the double input architecture outperforms the other neural networks. It has also been proven that, the enhanced classifier has a better detection rate than many of the most popular antivirus products with access in millions of samples.



(a) ROC Curve.



(b) ROC Curve zoomed.

Figure 5.4: ROC Curve results.

# Chapter 6

## Conclusion

In this thesis, the use of a double input deep neural network for static malware detection is proved to be viable and has the potential for further study. Experiments show that by combining different forms of a sample, can be efficient and could slightly improve the detection rate. Basic encoding techniques were used for file vectorization that can effectively summarize large files for classification. The importance of the availability of a large dataset in such domains cannot be overlooked. Therefore, everything needed to reproduce this research is open-source and could be found in GitHub[3].

Further research in this area will be required to establish how efficient double input neural networks can be as classifiers. More complicated layers could be used to perform malware detection as long as samples with more complexity are collected. Advanced layers could lead to overfit of the network at a really early stage due to their complexity. Better feature selection techniques or more advanced image conversion algorithms could be applied. A larger feature space could be used by changing the dimensions of the images to 256x256 or extracting more raw bytes from the executable.

# Bibliography

- [1] “Malshare project.” [Online]. Available: <https://malshare.com/>
- [2] “VirusTotal.” [Online]. Available: <https://www.virustotal.com/gui/>
- [3] P. Bellonias, “Malware detection using machine learning: A double input architecture.” [Online]. Available: <https://github.com/pb96/tuc-thesis>
- [4] Microsoft, “Fakerean,”  
[https://www.microsoft.com/en-us/wdsi/threats/  
malware-encyclopedia-description  
?Name=Win32/FakeRean&threatId=.](https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Win32/FakeRean&threatId=)
- [5] —, “Dontovo.a,”  
[https://www.microsoft.com/en-us/wdsi/threats/  
malware-encyclopedia-description?Name=TrojanDownloader  
:Win32/Dontovo.A&threatId=-2147342037.](https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=TrojanDownloader:Win32/Dontovo.A&threatId=-2147342037)
- [6] Wikipedia, “Portable executable 32 bit structure.” [Online]. Available: [https://en.wikipedia.org/wiki/Portable\\_Executable](https://en.wikipedia.org/wiki/Portable_Executable)
- [7] WikiPedia, “Gnu debugger,”  
[https://en.wikipedia.org/wiki/GNU\\_Debugger.](https://en.wikipedia.org/wiki/GNU_Debugger)
- [8] M. Al-Asli and T. A. Ghaleb, “Review of signature-based techniques in antivirus products,” in *International Conference on Computer and Information Sciences (ICCIS)*, Sakaka, Saudi Arabia, 2019, pp. 1–6.
- [9] G. Bonfante, M. Kaczmarek, and J.-Y. Marion, “Control flow graphs as malware signatures,” in *International Workshop on the Theory of Computer Viruses*, Nancy, France, May 2007, pp. 1–7.

- 
- [10] W. Wong and M. Stamp, “Hunting for metamorphic engines,” *Journal in Computer Virology*, vol. 2, no. 3, pp. 211–229, Dec. 2006.
- [11] A. Moser, C. Kruegel, and E. Kirda, “Limits of static analysis for malware detection,” in *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, Miami Beach, FL, 2007, pp. 421–430.
- [12] M. Dalla Preda, M. Christodorescu, S. Jha, and S. Debray, “A semantics-based approach to malware detection,” *ACM Trans. Program. Lang. Syst.*, vol. 30, Aug. 2008.
- [13] P. O’Kane, S. Sezer, and K. McLaughlin, “Obfuscation: The hidden malware,” *IEEE Security Privacy*, vol. 9, no. 5, pp. 41–47, 2011.
- [14] J. Oberheide, M. Bailey, and F. Jahanian, “Polypack: An automated online packing service for optimal antivirus evasion,” in *Proceedings of the 3rd USENIX Conference on Offensive Technologies*, Montreal, Canada, 2009, p. 9.
- [15] Wikipedia, “Hex dump,”  
[https://en.wikipedia.org/wiki/Hex\\_dump](https://en.wikipedia.org/wiki/Hex_dump).
- [16] M. M. Masud, L. Khan, and B. Thuraisingham, “A hybrid model to detect malicious executables,” in *2007 IEEE International Conference on Communications*, Glasgow, Scotland, 2007, pp. 1443–1448.
- [17] Z. Fuyong and Z. Tiezhu, “Malware detection and classification based on n-grams attribute similarity,” in *IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing*, vol. 1, Guangzhou, China, 2017, pp. 793–796.
- [18] S. Jain and Y. K. Meena, “Byte level n-gram analysis for malware detection,” in *Computer Networks and Intelligent Computing*, K. R. Venugopal and L. M. Patnaik, Eds., Jan. 2011, pp. 51–59”.

- 
- [19] R. K. Shahzad, N. Lavesson, and H. Johnson, “Accurate adware detection using opcode sequence extraction,” in *6th International Conference on Availability, Reliability and Security*, Vienna, Austria, 2011, pp. 189–195.
- [20] I. Santos, F. Brezo, X. Ugarte-Pedrero, and P. Bringas, “Opcode sequences as representation of executables for data-mining-based unknown malware detection,” *Information Sciences*, vol. 231, pp. 203–216, Aug. 2013.
- [21] A. Shabtai, R. Moskovitch, C. Feher, S. Dolev, and Y. Elovici, “Detecting unknown malicious code by applying classification techniques on opcode patterns,” *Security Informatics*, vol. 1, pp. 1–22, 2011.
- [22] E. Raff, R. Zak, R. Cox, J. Sylvester, P. Yacci, R. Ward, A. Tracy, M. Mclean, and C. Nicholas, “An investigation of byte n-gram features for malware classification,” *Journal of Computer Virology and Hacking Techniques*, pp. 1–20, Sep. 2016.
- [23] G. Amato, “Peframe,”  
<https://github.com/guelfoweb/peframe>.
- [24] M. N. A. Zabidi, M. A. Maarof, and A. Zainal, “Malware analysis with multiple features,” in *2012 UKSim 14th International Conference on Computer Modelling and Simulation*, Cambridge, UK, 2012, pp. 231–235.
- [25] Zongqu Zhao, “A virus detection scheme based on features of control flow graph,” in *2011 2nd International Conference on Artificial Intelligence, Management Science and Electronic Commerce (AIMSEC)*, Dengfeng, China, 2011, pp. 943–947.
- [26] M. Shankarapani, K. Kancharla, S. Ramammoorthy, R. Movva, and S. Mukkamala, “Kernel machines for malware classification and similarity analysis,” in *The 2010 International Joint Conference on Neural Networks (IJCNN)*, Barcelona, Spain, 2010, pp. 1–6.

- 
- [27] S. Tang, “The detection of trojan horse based on the data mining,” in *2009 Sixth International Conference on Fuzzy Systems and Knowledge Discovery*, vol. 1, Tianjin, China, 2009, pp. 311–314.
- [28] X. Ugarte-Pedrero, I. Santos, P. G. Bringas, M. Gastesi, and J. M. Esparza, “Semi-supervised learning for packed executable detection,” in *5th International Conference on Network and System Security*, Milan, Italy, 2011, pp. 342–346.
- [29] T. Wang, C. Wu, and C. Hsieh, “Detecting unknown malicious executables using portable executable headers,” in *5th International Joint Conference on INC, IMS and IDC*, Seoul, South Korea, 2009, pp. 278–284.
- [30] “Vx heaven website.” [Online]. Available: <http://vxheaven.0l.wtf/>
- [31] 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*, Pittsburgh, Pennsylvania, USA, 2011.
- [32] M. Ahmadi, D. Ulyanov, S. Semenov, M. Trofimov, and G. 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*, New Orleans, Louisiana, USA, 2016, p. 183–194.
- [33] Microsoft, “Pe format,”  
<https://docs.microsoft.com/en-us/windows/win32/debug/pe-format>.
- [34] QuarksLab, “Lief module,”  
<https://github.com/lief-project/LIEF>.
- [35] “Numpy package.” [Online]. Available: <https://numpy.org/>
- [36] D. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *International Conference on Learning Representations*, Dec. 2014.

- [37] “Keras library.” [Online]. Available: <https://keras.io/>