August 2019

Static Malware Detection using Deep Neural Networks on Portable Executables

Piyush Aniruddha Puranik

STATIC MALWARE DETECTION USING DEEP NEURAL NETWORKS ON PORTABLE EXECUTABLES

By

Piyushaniruddha Puranik

Bachelor of Engineering - Computer
Savitribai Phule Pune University, India
2015

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science in Computer Science

Department of Computer Science
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
August 2019

UNLV | GRADUATE COLLEGE

Thesis Approval

The Graduate College
The University of Nevada, Las Vegas

Date

This thesis prepared by

Piyushaniruddha Puranik

entitled

Static Malware Detection Using Deep Neural Networks on Portable Executables

is approved in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science
Department of Computer Science

Justin Zhan, Ph.D.
Examination Committee Chair

Hal Berghel, Ph.D.
Examination Committee Member

Kazem Taghva, Ph.D.
Examination Committee Member

Tiberio Garza, Ph.D.
Graduate College Faculty Representative

Kathryn Hausbeck Korgan, Ph.D.
Graduate College Dean

Abstract

There are two main components of malware analysis. One is static malware analysis and the other is dynamic malware analysis. Static malware analysis involves examining the basic structure of the malware executable without executing it, while dynamic malware analysis relies on examining malware behavior after executing it in a controlled environment. Static malware analysis is typically done by modern anti-malware software by using signature-based analysis or heuristic-based analysis.

This thesis proposes the use of deep neural networks to learn features from a malware's *portable executable* (PE) to minimize the occurrences of false positives when recognizing new malware. We use the EMBER dataset for training our model and compare our results with other known malware datasets. We show that using a simple deep neural network for learning vectorized PE features is not only effective, but is also less resource intensive as compared to conventional heuristic detection methods. Our model achieves an Area Under Curve (AUC) of 99.8% with 98% true positives at 1% false positives on the Receiver Output Characteristics (ROC) curve. We further propose the practical implementation of this model to show that it can potentially compliment or replace conventional anti-malware software.

Acknowledgements

I would like to thank Dr. Justin Zhan for his help in guiding me towards picking a topic and completion of this thesis. I have always been interested in data science, and I am grateful that under the guidance of Dr. Zhan, I was able to achieve my goals and learn the fundamentals of quality research.

I am grateful to Dr. Berghel for his insights into the ethics of computing, and for putting up with all trouble I have caused him over the course of my research for this thesis. I would like to thank Dr. Taghva for introducing me to probabilistic models and the real world problems in data science. His classes have always motivated me towards exploring more in this field. I would like to thank Dr. Tiberio Garza for being a part of my thesis committee and for investing his time towards the completion of this thesis. For all the professors who've made learning at University of Nevada, Las Vegas a pleasure and have inspired me to learn, I am eternally grateful.

My mother, Vijaya Puranik, has always been a pillar of strength for me and has taught me how to face life unwavering in times of turmoil and difficulty. I want to thank, Saniya Puranik, my sister, for her sound and logical advice which has kept me grounded through my time here at UNLV.

Lastly, I would like to thank Amruta, my fiancee, who has supported me throughout all the difficult times in my life and has never stopped believing in me.

Piyushaniruddha Puranik

University of Nevada, Las Vegas

August 2019

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

The concept of malware detection mainly deals with analyzing executable files to establish malicious intent. Since the advent of anti-malware software, we have seen a rise in sophisticated malware which are specifically designed to circumvent this software. This in turn has spearheaded research into more advanced detection techniques. Malware analysis or malware detection can be performed in two ways: statically, and dynamically.

Static Malware Detection: Static malware detection is the process of analyzing a binary file without executing it. This can involve disseminating the file entirely and examining every component, using a disassembler to reverse engineer it, or converting it into assembly code to examine its flow [SH12]. It can also extend to the original source code of the software if available [ESKK12]. This is usually the first line of defense against malware used by all anti-malware software.

Dynamic Malware Detection: Dynamic malware detection uses behavior analysis while a malware is running to determine malicious intent. Usually, this is done in a sandbox environment to ensure that the executable does not cause any harm to the target system. This form of analysis is usually resource intensive and can be circumvented in various ways. Debuggers can also be used to analyze system calls or other behavioural patterns which cannot otherwise be detected using black box testing [Ker16].

For the scope of this thesis, we will be focusing only on static malware detection.

Machine learning has long been used for classifying data with complex characteristics that cannot be easily determined using mathematical functions. Deep neural networks are used today for various different applications including (but not limited to) data classification, data prediction, image recognition, natural language processing, and so on. This versatility of neural networks is perfect for something like big data where large amounts of data is available, but processing it to obtain a specific result is computationally expensive.

Until recently, the lack of availability of labeled datasets for supervised learning had slowed down progress in using machine learning or deep learning for malware detection. Igor Santos et al. proposed OPEM [SDB$^+$13] as static-dynamic approach to use machine learning for detecting unknown malware. They proposed analyzing operational codes obtained from disassembly of executables and analyzing their execution trace to determine malicious intent. Similarly, a dynamic malware detection framework for Android called DroidDolphin managed to achieve 86.1% accuracy [WH14] using dynamic malware analysis. Both methods are generally computationally expensive and suffer from limited availability of labeled data.

1.1 Objective

The main objective of this thesis is to design and evaluate a deep neural network for statically analyzing portable executable files to classify them as malicious or benign. For this purpose we use the EMBER dataset [AR18] containing data extracted from portable executables of known malicious and benign files. We use the *hashing-trick* [WDA$^+$09] for creating a mathematical summary of the features and standardizing our input vector.

We proceed to compare our model to similar models proposed previously for tackling static analysis of malware. We show a model simulating practical implementation of such a model for further research. We provide complete sources for reproducing the proposed model and the derived results.

1.2 Outline

- Chapter 1 introduces the concepts covered in this thesis.
- Chapter 2 talks about previous work done in the fields of static malware analysis and the machine learning approach used in malware detection.
- Chapter 3 describes the various aspects of portable executable files which need to be studied before understanding the proposed model. It briefly covers the dataset used for our model and how it relates the portable executable file format.
- Chapter 4 describes in detail the steps and processes involved in implementing our model along with the entire structure of the final model.

- In Chapter 5 we discuss the experiments conducted on our model and their implications in the real world. Resources to access the source code for the model and all experiments conducted are included in the chapter.

- Chapter 6 summarizes the contents of the thesis, the model, and areas of research that have not been covered in this thesis. It also talks about potential gaps in this thesis and research that could be done to bridge these gaps.

Chapter 2

Literature Review

In this section, we cover work published in using machine learning for malware detection. Some implementations are similar to the ones covered in this thesis, but are not reproducible due to the lack of availability of the data set used, or the use of proprietary frameworks for obtaining results. We also cover some related work in this field which deals with malware detection on other platforms using static as well as dynamic analysis of files.

2.1 Static Malware Analysis

There are various challenges involved with static malware analysis. Most of these problems can be solved by using dynamic malware analysis such as file corruption during runtime, code obfuscation or encrypted binary executable files. Below, we explore some of these problems and the shortcomings of semantic-analyzers in solving them.

2.1.1 Signature Avoidance

Typically, anti-virus software use a signature based method to detect malware. The instructions present in the malware executable are parsed to obtain a unique signature identifying the malware which is then compared to a large database of known malware signatures [EMO12, OSM11]. Bonfante *et al.* proposed a control flow graph method to combat this problem [BKM07]. They used a graph with nodes for all commonly used assembly instructions, and then used a reduced version of this graph as a signature to classify malware. According to their tests, this form of detection resulted in better overall detection accuracy when the graphs were larger (for larger executables).

2.1.2 Code Obfuscation

Static malware analysis has mainly been studied from the perspective of semantic analysis and source code analysis for classification. Moser *et al.* proposed a method for obfuscating code from semantic analysis [MKK07] simply by using opaque constants to obscure program control flow. This highlights a significant flaw in static malware analysis techniques present today with semantic-aware analysis wherein semantic analysis can be 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. The introduction of a NP-hard algorithm to determine the value of certain constants in the code was also discussed in this paper. For example, implementing a 3SAT problem in code such that the input variables to this section code will always return a static value (say 0). This means that the program will always generate a value of 0 during run time when any variable is assigned to the 3SAT algorithm. Although this is easy to determine by a human reading the code, it is very difficult for a semantic-aware analyzer to determine all possible outputs of this algorithm and finally determine that the output of this is always 0, since the algorithm cannot be computed in polynomial time. Code obfuscation using encryption on binary files multiple times and then bundling a tool for decryption was discussed by Christodorescu and Jha [CJ06]. This form of obfuscation is easy to catch during runtime by analyzing the decrypted file in memory, but it is difficult to determine the level of encryption of the file without first decryption and analyzing it dynamically.

A semantics based approach which proposed a metric for gauging the similarity between original malware code and obfuscated malware code was proposed by Preda *et al.* [PCJD07]. It also discussed methods to detect the inclusion of constant obfuscation (by adding NP-hard computation or similar such methods) in malware code, NOP insertion, command substitution, and variable renaming. However, the practical implementation of this approach has not been fully realized. There are plenty of dynamic malware detection methods including call graph analysis [EMO12] and identifying behaviour based on triggers [BHL^{+}08]. However, these methods are computationally expensive and require a sandbox infrastructure where malware can be safely executed and analyzed.

2.1.3 Software Packing

File packing is a common technique used when bundling large software in a small, compact package [OSM11]. Such packaging techniques usually involve some form of encryption which can potentially prevent easy identification of malware. One such tool called `PolyPack` [OBJ09] is specifically designed to prove that packers are an effective method of evading anti-virus and anti-malware software. They provide 10 packers which all independently pack the data provided to them, and then scan the packed data with 10 well known anti-virus scanners. The packer with the best result is picked. Their studies established that this improved evasion rates by 2.58 times against most anti-virus software.

2.2 Machine Learning for Malware Detection

The fact that machine learning performs better with larger datasets is well established [BB01]. Several studies have been published which use machine learning for malware classification. Various methods such as dynamic analysis of system calls [KZWE16], registry access monitoring [HSKS03], hidden Markov model based analysis [AMS09] have been proposed for dynamic malware analysis.

Kolter and Maloof [KM04] proposed the use of *n-grams* by combining 4 byte sequences to produce approximately 255 million distinct *n-grams*. This paper proposed the use of a probabilistic approach for determining which features were relevant and used the top 500 *n-grams* for analysis. The paper proposed the use of Naive Bayes, Support Vector Machine (SVM), and J48 decision tree for analyzing their data. Data used for analysis was primarily sourced from Sourceforge and VX Heavens (actual data not disclosed) with 1971 benign executables and 1651 malicious executables tested. The small sample set used in this research, and the fact that the exact dataset used by the authors is not available, it is difficult to ascertain the accuracy of these results when used with larger datasets. A similar study was done by Bagga [Bag17] using this approach with the Microsoft Malware Classification [RRF$^+$18], which is an arguably large dataset. However, this study focused on the malware classification problem rather than the malware detection problem.

Raman, from the Product Incident Response Team at Adobe Systems Inc. proposed a method to classify malware by extracting seven least correlated features from portable executables [R$^+$12]. The features extracted were *DebugSize, ImageVersion, IatRVA, ExportSize, ResourceSize, VirtualSize, NumberOfSections.* A dataset containing 100,000 malicious executables and 16,000 benign executables was used for experimentation. Various models were tested using this data. Amongst the models tested, the J48 decision tree [Qui93] obtained the best results: a true positive rate of 0.986 with a false positive rate of 0.057. The resulting trained model was released as a free tool for malware classification, but the dataset was not published to perform any form of comparative research. Anderson and Roth further tested this trained model with the EMBER dataset [AR18] and found that it exhibited a false positive rate of 0.53 and a false negative rate of 0.08.

A dynamic malware classification model using deep neural networks called `MtNet` was proposed by Huang and Stokes in 2016 [HS16]. The dataset used for this study was provided by Microsoft Corporation containing 6.5 million sample files. From this dataset, 2.85 million malicious and 3.65 million benign files were extracted. Training features were extracted during file execution at runtime consisting of mainly two types of data: system function calls and null-terminated objects. Feature selection was performed using mutual information proposed by Manning *et al.* [MRS10] to get a total of 50,000 input features. The final goal was to classify malware first as benign or malicious, and then classify the malicious malware into one of 100 known malware families. The ReLU activation function was used along with dropout layers added for better model performance. Although this model shows impressive results of under 0.07% false positive rates, the lack of availability of the test data set and the model code used for testing makes reproducing these results impossible.

Echo state network and recurring neural network based malware classifiers have also been tested for dynamic analysis of malware by Pascanu *et al.* [PSS^{+}15]. Their research established the use of an echo state network based recurrent model with the sigmoid (logistic regression) activation function for dynamic analysis of malware. The exact input vector was not disclosed, however it was derived from the API calls performed by files during runtime execution. The model achieved a true positive rate of 0.983 with a false positive rate of 0.001. The authors acknowledge that the dataset used in this research was sourced internally, and is not publicly available. The purpose of this research was to establish that recurrent neural networks can be used for dynamic malware analysis. However, due the the dataset not being available, and the steps required to reproduce the proposed model not provided, it is difficult to verify these results and conduct further research based off of it.

2.2.1 Feature Selection

Machine learning is very sensitive to the feature set being used for training. Various studies have established certain features to be beneficial for effective training of machine learning based malware classifiers. We discuss some of these approaches.

Divandari *et al.* proposed extracting opcode data from files and using a Markov Blanket approach to summarizing the feature set [DPJ15]. Since opcodes themselves are a significant portion of executables, they have been considered as reliable features for malware detection [Bil07]. The proposed model uses a Hidden Markov Model (HMM) for malware classification.

The byte histogram approach proposed by Saxe and Berlin in their research [SB15] introduced a format-agnostic method of extracting features from a file. This method is an innovative approach to extracting byte information as features from a file without requiring information about the actual function of those bytes. It proposes to extract a histogram of all byte values present in the binary file along with a 2 dimensional byte-entropy histogram to establish an understanding of potential encryption or compression used in the file. We use this method in our model to complement the header extraction method such that we achieve high accuracy without the high overheads required for vectorizing all bytes in the portable executable.

The feature hashing trick proposed by Weinberger *et al.* [WDA+09] has been frequently cited and used for machine learning models. The input vector for most machine learning based models is static and cannot be increased in size depending upon input size. Therefore, we need a method to effectively summarize large input features into a static size which is more manageable for training. The feature hashing trick proposes a method to effectively reduce the dimensionality of data such that it still sufficiently represent the original intended data, but offer linear separable features for training a model effectively.

2.2.2 Boosted Decision Trees and Artificial Neural Networks

Decision tree have been around for a long time. However, with the recent advancement in the boosting method for decision tree models, they have proved to be similar or better in performance than artificial neural networks. They are relatively easier to tune and work well with large number of variables [RYZ+05]. With the advent of AdaBoost, boosting of decision tree models has managed to go from binary classification to multi-category classification [HRZZ09, Sch03]. This spurred the use of boosted decision tree based models as alternatives for artificial neural networks. The model we propose in this thesis is compared to an existing boosted decision tree model for the same dataset that we use for our model. Caruana and Niculescu-Mizil established in their paper [CNM06] that state vector machines, boosted decision trees and neural nets have comparable performance in most scenarios with variance mainly limited to hyper-parameter tuning.

Chapter 3

Background

This section briefly describes the portable executable (PE) file format and its header. We discuss the methodology used in extracting data from the portable executable file and generating input for our model. Section summarizes the structure of the EMBER dataset used for training our model. The methods used for hashing and standardizing the data are summarized in section.

3.1 Portable Executable Format

The portable executable (PE) format (Figure 3.1) was introduced by Microsoft with the Windows NT 3.1 operating system. Since its inception, it has seen several improvements to incorporate it into newer versions of Windows. Unix uses the ELF format which is analogous to the Windows PE format.

The scope of this thesis is limited to Windows executables since data available for malware running on Unix based operating systems is limited. However, the COFF header found in PE files is common to both Unix and Windows environments [Kat93]. Our proposed model analyzes features extracted from PE files to determine whether the file is malicious or benign. 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 an 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. [Cen19]

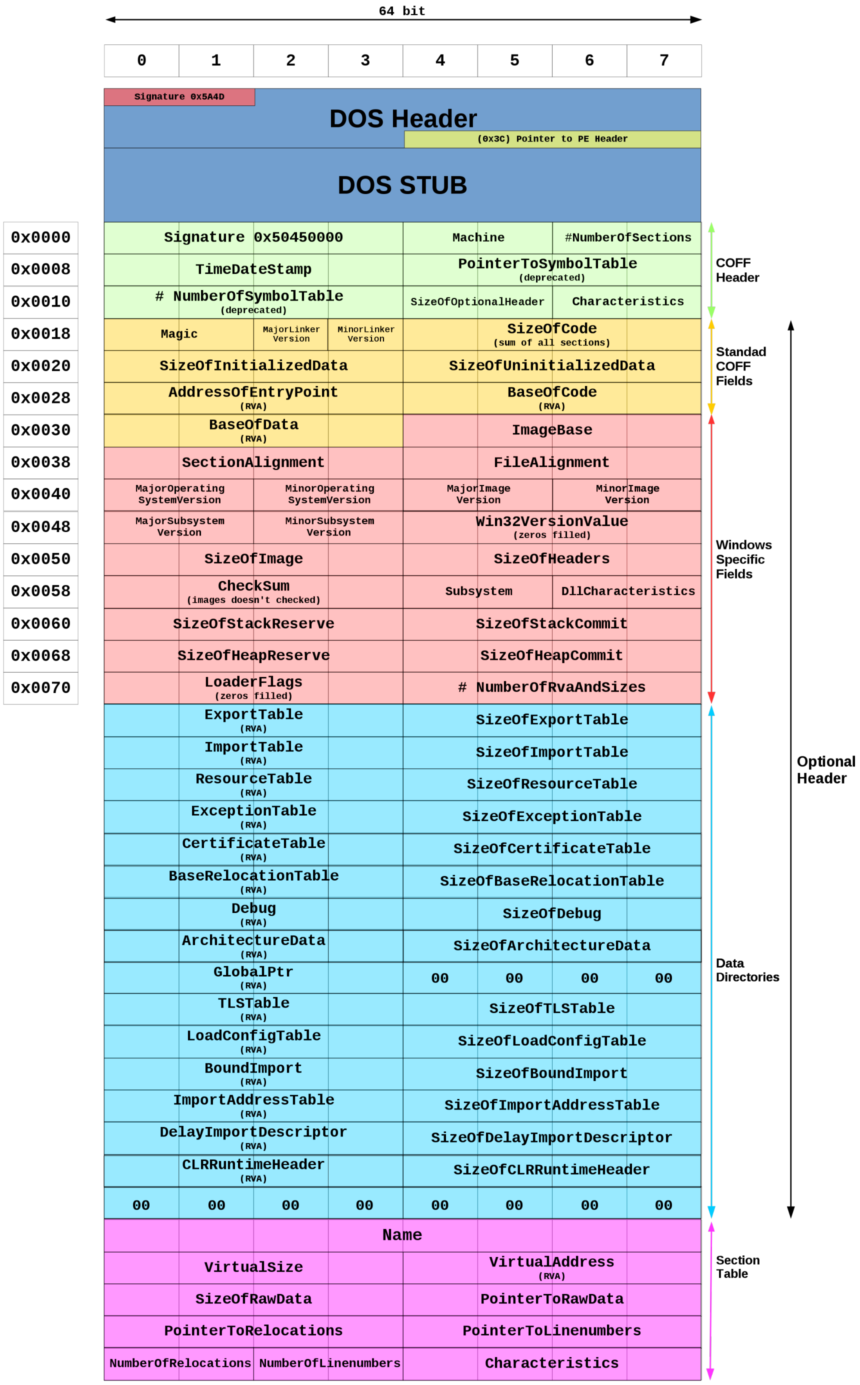


Figure 3.1: PE File Format [Com16]

3.1.2 COFF Header

The Common Object File Format (COFF) header exists right after the MS-DOS Stub. The COFF Header structure is defined in Table 3.1. All possible values for the *Machine* field and *Characteristics* field in the COFF header are defined in Table 3.2 and Table 3.3 respectively.

Offset	Size	Field	Description
0	2	Machine	Identifies the target machine that the executable can run on. Refer to Table 3.2
2	2	NumberOfSections	Size of the section table. (follows the header table)
4	4	TimeDateStamp	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. Refer to Table 3.3.

Table 3.1: COFF Structure [Cen19]

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

3.1.3 Optional Header

Files which are considered as executables (images) have an additional optional header. This header provides information to the loader present in the operating system which is responsible for handling execution of executable files. Although this header is required for executable files, it can also be present in *object* files. Optional headers in object files serve no purpose except to increase file size.

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+ as is shown in table 3.4.

PE32+ executables allow 64-bit memory address space, but can be no more than 2 gigabytes in size. The optional header is split into 3 major parts defined in Table 3.5.

Standard fields in the optional header are defined for every COFF implementation (Windows and Unix). Following is a summary of the information contained in the section:

- Magic number indicating whether the file is a normal executable (0x10B), a ROM image (0x107), or a PE32+ executable (0x20B).

- Linker version to be used for this PE file.

Constant	Value	Description
IMAGE_FILE_MACHINE_UNKNOWN	0x0	Applicaple to any machine
IMAGE_FILE_MACHINE_AM33	0x1d3	Matsushita AM33
IMAGE_FILE_MACHINE_AMD64	0x8664	x64
IMAGE_FILE_MACHINE_ARM	0x1c0	ARM little endian
IMAGE_FILE_MACHINE_ARM64	0xaa64	ARM64 little endian
IMAGE_FILE_MACHINE_ARMNT	0x1c4	ARM Thumb-2 little endian
IMAGE_FILE_MACHINE_EBC	0xebc	EFI byte code
IMAGE_FILE_MACHINE_I386	0x14c	Intel 386 or equivalent
IMAGE_FILE_MACHINE_IA64	0x200	Intel Itanium processor family
IMAGE_FILE_MACHINE_M32R	0x9041	Mitsubishi M32R little endian
IMAGE_FILE_MACHINE_MIPS16	0x266	MIPS16
IMAGE_FILE_MACHINE_MIPSFPU	0x366	MIPS with FPU
IMAGE_FILE_MACHINE_MIPSFPU16	0x466	MIPS16 with FPU
IMAGE_FILE_MACHINE_POWERPC	0x1f0	Power PC little endian
IMAGE_FILE_MACHINE_POWERPCFP	0x1f1	Power PC with floating point support
IMAGE_FILE_MACHINE_R4000	0x166	MIPS little endian
IMAGE_FILE_MACHINE_RISCV32	0x5032	RISC-V 32-bit address space
IMAGE_FILE_MACHINE_RISCV64	0x5064	RISC-V 64-bit address space
IMAGE_FILE_MACHINE_RISCV128	0x5128	RISC-V 128-bit address space
IMAGE_FILE_MACHINE_SH3	0x1a2	Hitachi SH3
IMAGE_FILE_MACHINE_SH3DSP	0x1a3	Hitachi SH3 DSP
IMAGE_FILE_MACHINE_SH4	0x1a6	Hitachi SH4
IMAGE_FILE_MACHINE_SH5	0x1a8	Hitachi SH5
IMAGE_FILE_MACHINE_THUMB	0x1c2	Thumb
IMAGE_FILE_MACHINE_WCEMIPSV2	0x169	MIPS little-endian WCE v2

Table 3.2: COFF: Machine Types [Cen19]

Flag	Value	Description
IMAGE_FILE_RELOCS_STRIPPED	0x0001	The file must be loaded at its preferred base address because it does not allow base relocation.
IMAGE_FILE_EXECUTABLE_IMAGE	0x0002	Set for valid files. Linker error if this is not set.
IMAGE_FILE_LINE_NUMS_STRIPPED	0x0004	Deprecated. Set to zero.
IMAGE_FILE_LOCAL_SYMS_STRIPPED	0x0008	Deprecated. Set to zero.
IMAGE_FILE_AGGRESSIVE_WS_TRIM	0x0010	Obsolete for Windows 2000 and later. Set to zero.
IMAGE_FILE_LARGE_ADDRESS_ AWARE	0x0020	Capable of handling addresses more than 2GB.
	0x0040	Reserved.
IMAGE_FILE_BYTES_REVERSED_LO	0x0080	Little Endian. Deprecated. Set to zero.
IMAGE_FILE_32BIT_MACHINE	0x0100	Machine uses 32-bit architecture.
IMAGE_FILE_DEBUG_STRIPPED	0x0200	File does not have debug information.
IMAGE_FILE_REMOVABLE_RUN_ FROM_SWAP	0x0400	Copy the image to memory if it is on removable media.
IMAGE_FILE_NET_RUN_FROM_SWAP	0x0800	Copy the image to memory if it is on network media.
IMAGE_FILE_SYSTEM	0x1000	System File
IMAGE_FILE_DLL	0x2000	DLL File. Cannot be executed.
IMAGE_FILE_UP_SYSTEM_ONLY	0x4000	Only support uniprocessor machine.
IMAGE_FILE_BYTES_REVERSED_HI	0x8000	Big Endian. Deprecated. Set to zero.

Table 3.3: COFF: Attribute Flags Available [Cen19]

Magic number	PE format
0x10b	PE32
0x20b	PE32+

Table 3.4: Optional Header Magic Number [Cen19]

Offset (PE32/PE32+)	Size (PE32/PE32+)	Header part	Description
0	28/24	Standard fields	Common for Windows and Unix COFF implementations.
28/24	68/88	Windows-specific fields	Defines windows specific features.
96/112	Variable	Data directories	Address and size of special tables used by OS.

Table 3.5: Optional Header Parts [Cen19]

- Size of the code section. Code section refers to the text section of a PE file which contains the actual software that will be run when the file is executed. There can be multiple such code sections within the file, in which case, the header field will indicate the total size of all the code sections combined. Code sections are also referred to as the *.text* section of a PE file.

- Size of initialized and uninitialized data contained with the file. This is also referred to as the *.data* section of a PE file.

- Address of the entry point of the file. This is where the instruction pointer will start when the PE file is loaded into memory. [Cen19]

Windows specific fields contain certain information required specifically for Windows environments. It contains operating system version, image version (for example Word version 8.0), 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. [Kat93]

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. [Cen19]

3.1.4 Section Table

Every section in the PE file contains a section header which is 40 bytes in size. This defines the name of the section, virtual size, number of lines, and various pointers (lines, raw data, relocations, etc.) [Cen19].

Besides the sections described above, the PE file contains the software executable code. There are a few other miscellaneous sections that can be included depending upon the file, but that is beyond the scope of this thesis.

3.2 Training Dataset

Since our model analyzes PE files, the first challenge was to find a dataset that provides PE files labeled to be malicious or benign. Up until the EMBER dataset was released in April 2018, it was difficult to find datasets which classify malware as malicious or benign. This dataset is 9.1 GB in size and provides 900K training samples with 300K malicious, 300K benign, and 300K unlabeled

Offset	Size	Field	Description
0	8	Name	Name of the section represented as an 8-byte UTF-8 string.
8	4	VirtualSize	Size of the section in memory.
12	4	VirtualAddress	Refers to the address of the first byte when loaded to memory.
16	4	SizeOfRawData	Size of the uninitialized data on disk.
20	4	PointerToRawData	File pointer to the first page of the section.
24	4	PointerToRelocations	Set to zero for executable files.
28	4	PointerToLinenumbers	Deprecated. Set to zero.
32	2	NumberOfRelocations	Set to zero for executable files.
34	2	NumberOfLinenumbers	Deprecated. Set to zero.
36	4	Characteristics	Characterstic Flags

Table 3.6: Section Header [Cen19]

samples. It also contains 200K test samples. The publishers of this dataset have also released the source code that they used for creating this dataset. We use this code as a basis for extracting features from binary files. [AR18]

The data is provided in JSON files in which every line is one sample. Each sample contains parsed features, which are essentially features extracted from the PE header described in section 3.1, and format agnostic features as described below.

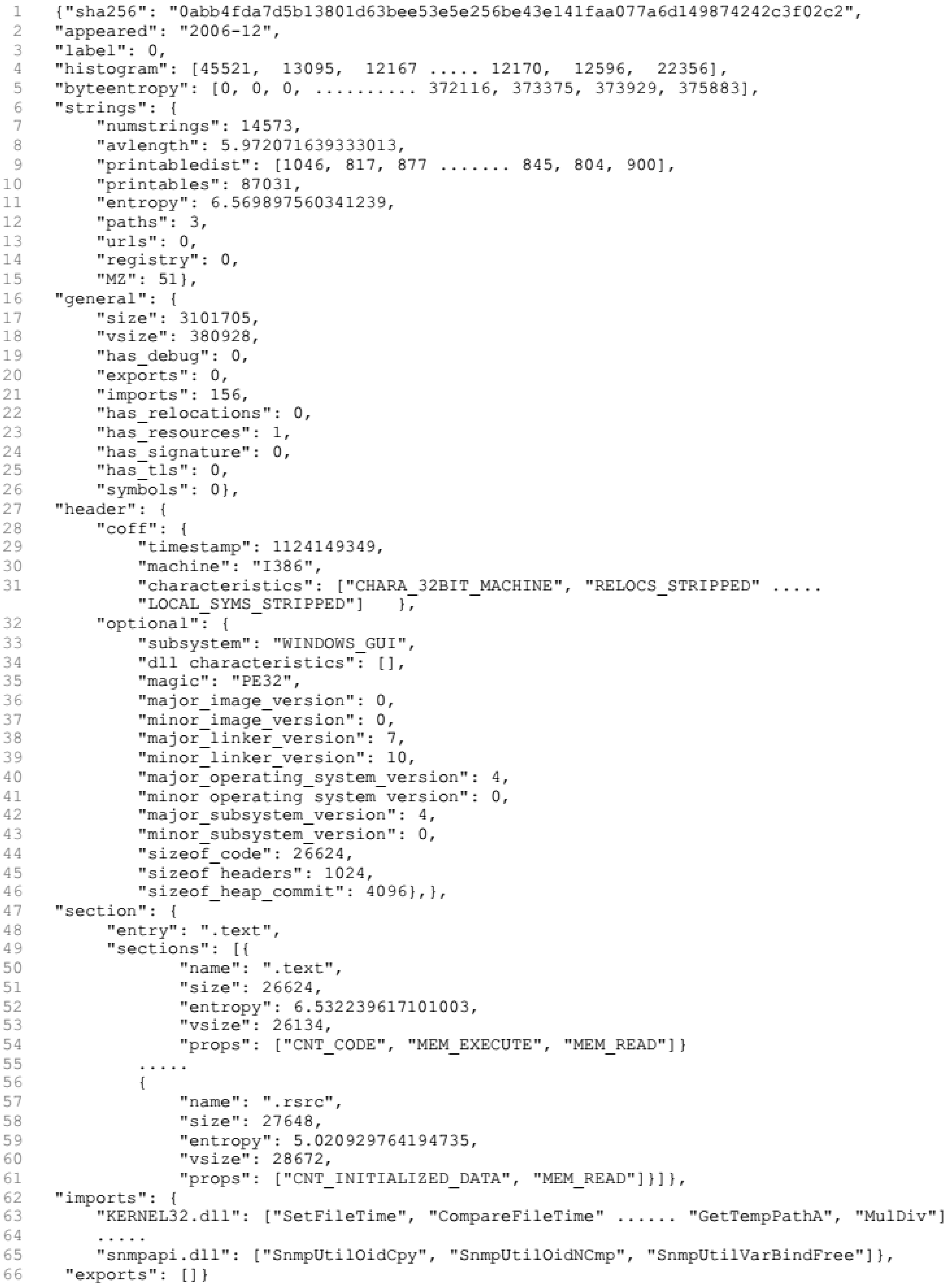
- SHA 256 Checksum.
- When was this file first seen.
- Label (0 for benign, 1 for malicious, -1 for unlabeled).
- Histogram of the raw bytes in the file.
- Entropy of the raw bytes in the file.

A summarized view of a sample from the training set is shown in Figure 3.2.

The inclusion of byte histogram and byte entropy histogram as seen in Figure 3.2 has previously been seen in [SB15] for including format agnostic features as part of the data set.

3.2.1 Byte Histogram

The byte histogram is essentially a vector of size 256 with each index representing the frequency of the corresponding byte value. For example, if there are 11203 occurrences of the byte value 200,



```
{"sha256": "0abb4fda7d5b13801d63bee53e5e256be43e141faa077a6d149874242c3f02c2",
"appeared": "2006-12",
"label": 0,
"histogram": [45521,  13095,  12167 ..... 12170,  12596,  22356],
"byteentropy": [0, 0, 0, .......... 372116, 373375, 373929, 375883],
"strings": {
    "numstrings": 14573,
    "avlength": 5.972071639333013,
    "printabledist": [1046, 817, 877 ....... 845, 804, 900],
    "printables": 87031,
    "entropy": 6.569897560341239,
    "paths": 3,
    "urls": 0,
    "registry": 0,
    "MZ": 51},
"general": {
    "size": 3101705,
    "vsize": 380928,
    "has_debug": 0,
    "exports": 0,
    "imports": 156,
    "has_relocations": 0,
    "has_resources": 1,
    "has_signature": 0,
    "has_tls": 0,
    "symbols": 0},
"header": {
    "coff": {
        "timestamp": 1124149349,
        "machine": "I386",
        "characteristics": ["CHARA_32BIT_MACHINE", "RELOCS_STRIPPED" .....
        "LOCAL_SYMS_STRIPPED"]    },
    "optional": {
        "subsystem": "WINDOWS_GUI",
        "dll characteristics": [],
        "magic": "PE32",
        "major_image_version": 0,
        "minor_image_version": 0,
        "major_linker_version": 7,
        "minor_linker_version": 10,
        "major_operating_system_version": 4,
        "minor operating system version": 0,
        "major_subsystem_version": 4,
        "minor_subsystem_version": 0,
        "sizeof_code": 26624,
        "sizeof headers": 1024,
        "sizeof_heap_commit": 4096},},
"section": {
     "entry": ".text",
     "sections": [{
            "name": ".text",
            "size": 26624,
            "entropy": 6.532239617101003,
            "vsize": 26134,
            "props": ["CNT_CODE", "MEM_EXECUTE", "MEM_READ"]}
        .....
        {
            "name": ".rsrc",
            "size": 27648,
            "entropy": 5.020929764194735,
            "vsize": 28672,
            "props": ["CNT_INITIALIZED_DATA", "MEM_READ"]}]},
"imports": {
    "KERNEL32.dll": ["SetFileTime", "CompareFileTime" ...... "GetTempPathA", "MulDiv"]
    .....
    "snmpapi.dll": ["SnmpUtilOidCpy", "SnmpUtilOidNCmp", "SnmpUtilVarBindFree"]},
 "exports": []}
```

Figure 3.2: JSON Sample

then the value present at index 200 of the histogram vector would be 11203 [AR18]. A detailed explanation of this process can be found in section 4.1.1.

3.2.2 Byte Entropy Histogram

To compute the byte entropy histogram, a window of size 2048 bytes is moved over the input bytes with a step size of 1024 bytes. The entropy of the entire 2048 byte window with the occurrence of each individual byte in the window is computed and stored as a pair. This results in a total of 2048 pairs. Bins of size 16 x 16 are used to quantize the entropy and byte values [SB15, AR18]. These values are normalized before training. Details of the byte histogram extraction process is given in section 4.1.1 and the the normalization process is described in section 4.2.

Chapter 4

Proposed Model

The model has 3 major components. They are as follows:

1. Feature Extraction and Hashing
2. Scaling and Normalization
3. Neural Network Classifier

We use Python for implementing our model due to its simplicity and flexibility of use in machine learning applications [Oli07]. We leverage the Nvidia CUDA architecture [NBG08] for high speed parallel computation using the Keras library [C^+15] for implementing our neural network model.

4.1 Feature Extraction and Hashing

There are two main components of the PE file we extract in order to train our model:

1. Parsed Information
2. Raw Byte Information

The EMBER dataset [AR18] used in our model provides a convenient module to extract the required data from any given PE file. We describe the processes involved in this module. In total, our model uses 2351 input vectors for classification.

4.1.1 Parsed Information

Every PE file contains header information as described in Section 3.1. These features are extracted in python using the LIEF library for parsing PE files [Tho17]. However, this information is not all numerical and not always the same size. Our model uses input vectors of fixed size for training. This means that all features extracted from the PE file must be brought to a standard size before they can be used for model training. To achieve this, we use the `FeatureHasher` module from the `scikit-learn` library [PVG+11] to implement the feature hashing trick [WDA+09] with a set number of bins per header feature. Five groups of features are extracted from the PE file.

General Information

General features obtained from the PE file include:

- Virtual size
- Imported functions
- Exported functions
- Presence of debug section
- Resources
- Relocations
- Number of Symbols

This is basic information obtained from the PE header [AR18].

Header Information

We obtain specific information from the headers present in the PE file. From the COFF header, we obtain the following information:

- Timestamp
- Target Machine (string)
- List of Image Characteristics (list of strings)

From the optional header we obtain the following:

- Target Subsystem (string)
- DLL Characteristics (list of strings)

- Magic Number
- Major Image Version
- Minor Image Version
- Linker Version
- System Versions
- Subsystem Versions
- Code Size
- Header Size
- Commit Size

Features involving strings are first converted to their byte representation and then parsed through the `FeatureHasher` to get 10 bins of summarized data. This ensures that the data is vectorized and is of a fixed size [AR18].

Imported Functions

Imported address tables from the optional headers along with the imported function sorted by library are extracted from the PE file. This data is summarized using the `FeatureHasher`. 256 bins are used for summarizing all unique libraries and 1024 bins are used to represent every `library:FunctionName` pair.

Exported Functions

All exported functions are extracted as strings and then summarized with 128 bins using the `FeatureHasher`.

Section Information

Section tables are extracted with the following information:

- Section Name
- Virtual Size

- Size
- Entropy
- Virtual Size
- Section Characteristics (list of strings)
- Entry Point

The hashing trick is used on the `Name:Value` pairs. The name being the section name, and the values being section size, section entropy and virtual size, each paired individually with a name. These pairs are summarized using the `FeatureHasher` with each value set allotted 50 bins. Section characteristics are captured separately using the same hashing trick as mentioned previously [AR18].

4.1.2 Raw Byte Information

Raw-byte information is included for platform agnostic analysis of the PE files. This means that malware not designed for a Windows environment can also potentially be classified. We do not test the accuracy of this model for non-Windows malware, so the effectiveness of this method for such cases is not discussed.

Byte information extracted from files is not dependent on the type of file. It is simply the representation of all bytes comprising the file. Since the size of these bytes is variable, we use a method proposed by Saxe and Berlin [SB15] to summarize this data. This method has also been implemented in the feature extraction module of the EMBER dataset [AR18].

Byte Histogram

Byte histogram is essentially the frequency of the occurrence of each byte value in a file. A byte can have a value from 0 to 255, which means a histogram of these bytes will contain the frequency of occurrence of 256 possible integer values. Algorithm 1 defines the steps involved in this process. A graphical representation of a sample byte histogram is shown in Figure 4.1.

Byte-Entropy Histogram

Assuming a window size of 2048 bytes and a stride of 1024 bytes, we compute the entropy histogram by sliding this window over the entire file, computing the Shannon entropy H of the 2048 byte window, and plotting the joint distribution of this window with every byte present in it. For all

Algorithm 1: Compute Byte Histogram

```
Result: List of size 256 containing Byte Histogram
Initialize $file$
Initialize list $count[256]$
while $file \neq EOF$ do
    $bytes \leftarrow file.read(bufsize)$
    foreach $value$ in $bytes$ do Increment $count[value]$

end
```

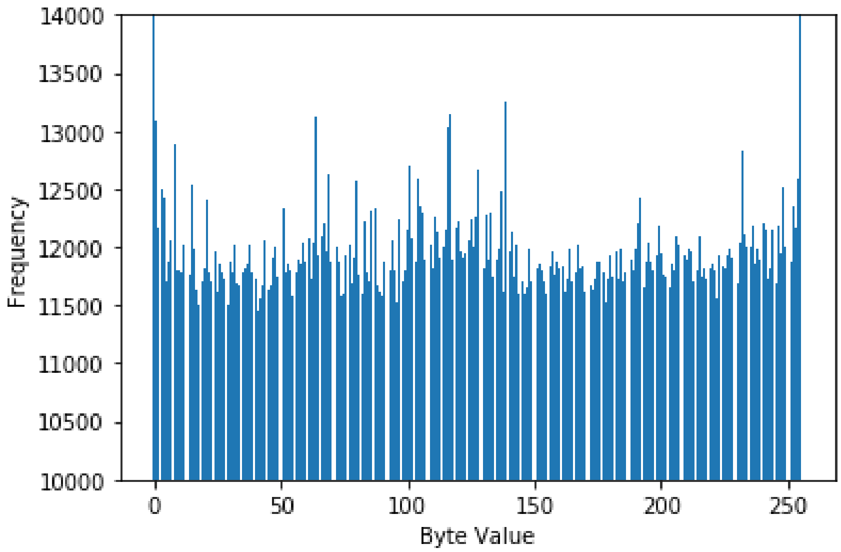


Figure 4.1: Example of Byte Histogram

bytes in the window such that $X = \{X_1, X_2....X_n\}$, where $P(X_i)$ is the probability of the occurrence of X_i in the proposed window, the entropy $H(X)$ of the window to the base b is calculated as:

$$H(X) = -\sum_{i=1}^{n} P(X_i) log_b P(X_i)$$

Here we will be calculating the entropy of the window to the base 2. Then we create a pair $P(H, X)$ where H is the entropy of the window, and X is the occurrence of the every byte in the window. An example of a byte entropy histogram is given in Figure 4.2.

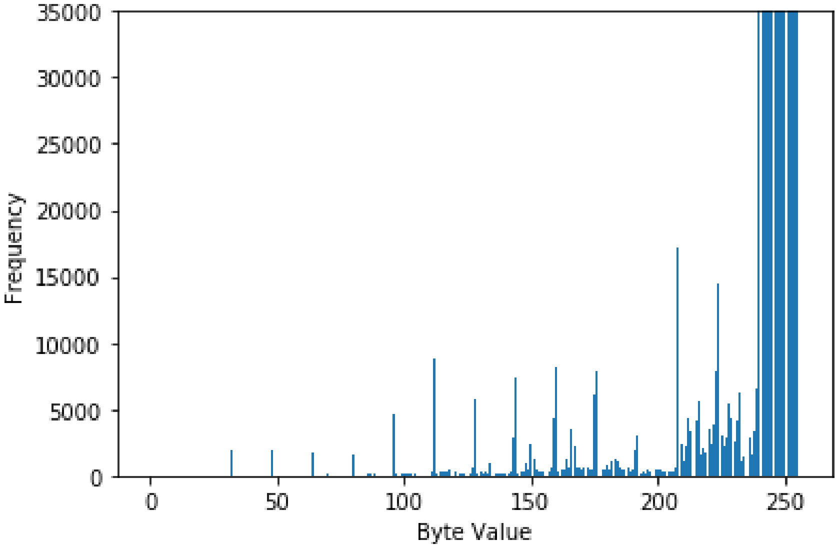


Figure 4.2: Example of Byte-Entropy Histogram

Printable Strings

Most PE files contain strings which are a valuable source of data. We are mainly interested in all printable characters. All strings containing more than 5 printable characters are extracted from the file. Extracted strings are categorized based on what they contain to create a group of features which provide a statistical summary of the string contents of the file. Only the number of elements contained in each category are used as model features. The categories are as follows:

- `C:\` (case sensitive) indicating a Windows path
- `http://` or `https://` indicating a URL
- `HKEY_` indicating a Windows registry key

This serves two purposes: it can reveal certain characteristics of a file which are not revealed from header information, and it protects the privacy of benign files since we are only creating a summary of the string data. [AR18].

4.2 Scaling and Normalization

After extracting features from the PE file, we obtain 2351 features per sample. However, these feature values are widely scattered with a large number of values being close to 0 and some of them being over 10^9 and some of them being under -10^4. This phenomenon is shown as a line graph for one of the test samples in Figure 4.3. A scatter plot of this phenomenon made it difficult to see the extreme points which made it necessary to plot a line graph.

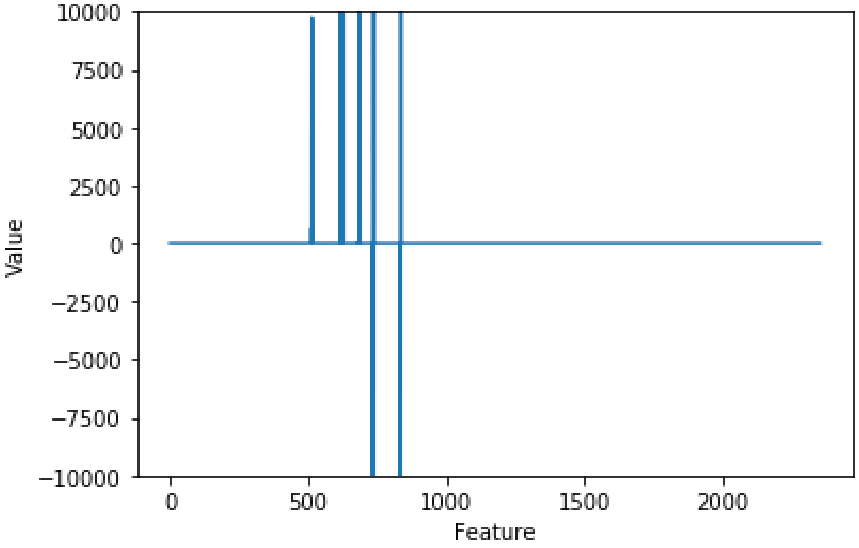


Figure 4.3: Line graph of raw sample

When attempting to train our model with this kind of data, we saw that the model would not converge and resulted in an AUC of 0.5. This was clearly due to the widely varying scales of the extracted features which needed some form of normalization before being used in our model. We use the statistical or Z-score normalization for this purpose [JS11].

Let the input sample be $X = \{X_1, X_2, X_3....X_{2351}\}$ where X_i represents the ith feature within the sample. The statistical normalization of the sample X for each feature X_i is performed using the equation:

$$X_i' = \frac{X_i - \bar{X}}{\sigma(X)}$$

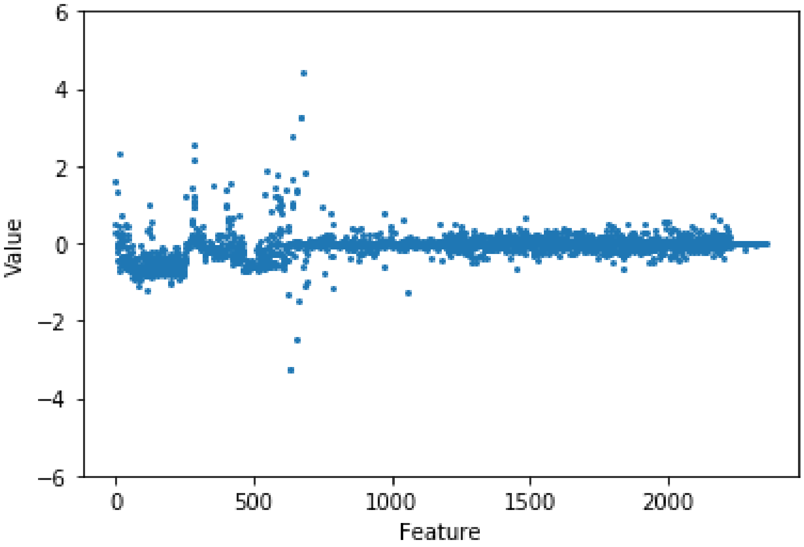


Figure 4.4: Scatter plot of normalized sample

Where $\bar{X}$ is the arithmetic mean of the sample X and $\sigma(X)$ is the standard deviation of the sample X. The arithmetic mean $\bar{X}$ of X is calculated using the equation:

$$\bar{X} = \frac{1}{2351} \sum_{i=1}^{2351} X_i$$

Where 2351 is the cardinality of the sample X. The standard deviation $\sigma(X)$ of X is calculated using the equation:

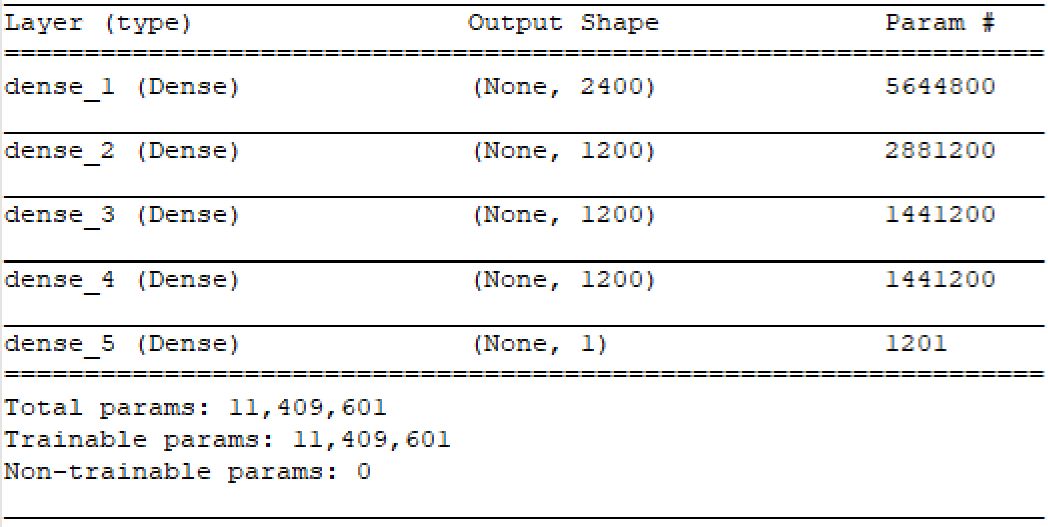
$$\sigma(X) = \sqrt{\frac{1}{2351} \sum_{i=1}^{2351} (X_i - \bar{X})^2}$$

A scatter plot of the normalized sample is given in Figure 4.4. We use the `StandardScalar` function provided in the `scikit-learn` library [PVG+11] for normalizing our samples. This function performs the exact same calculations as explained above.

4.3 Neural Network Classifier

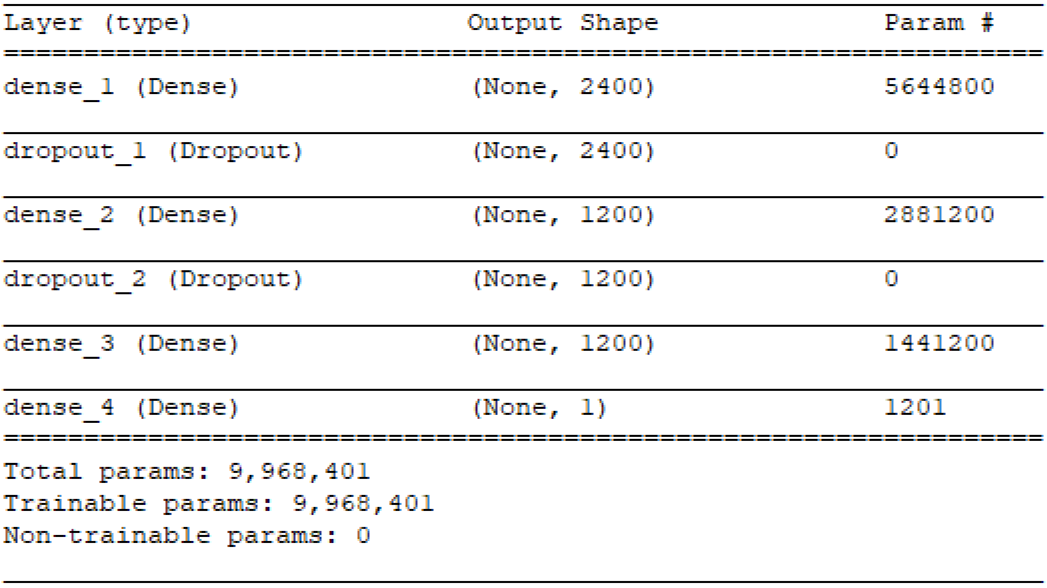
We use a deep neural network for analyzing our data. There were two neural networks constructed, one with dropout layers, and one without dropout layers. We tested the logistic activation function and the rectified linear unit (ReLU) activation function for both these networks. The Adam optimizer [KB14] implemented in the Keras library was used for gradient-based optimization of our classifiers. A summary of both neural networks are shown in Figure 4.5 and Figure 4.6, respectively.

The network without dropout layers contains 1 input layer which accepts an input vector of size 2351, 1 dense layer with 2400 neurons, 3 dense layers with 1200 neurons, and a binary output layer.



```
Layer (type)                 Output Shape              Param #   
=================================================================
dense_1 (Dense)              (None, 2400)              5644800   
_________________________________________________________________
dense_2 (Dense)              (None, 1200)              2881200   
_________________________________________________________________
dense_3 (Dense)              (None, 1200)              1441200   
_________________________________________________________________
dense_4 (Dense)              (None, 1200)              1441200   
_________________________________________________________________
dense_5 (Dense)              (None, 1)                 1201      
=================================================================
Total params: 11,409,601
Trainable params: 11,409,601
Non-trainable params: 0
_________________________________________________________________
```

Figure 4.5: Summary of neural network



```
Layer (type)                 Output Shape              Param #   
=================================================================
dense_1 (Dense)              (None, 2400)              5644800   
_________________________________________________________________
dropout_1 (Dropout)          (None, 2400)              0         
_________________________________________________________________
dense_2 (Dense)              (None, 1200)              2881200   
_________________________________________________________________
dropout_2 (Dropout)          (None, 1200)              0         
_________________________________________________________________
dense_3 (Dense)              (None, 1200)              1441200   
_________________________________________________________________
dense_4 (Dense)              (None, 1)                 1201      
=================================================================
Total params: 9,968,401
Trainable params: 9,968,401
Non-trainable params: 0
_________________________________________________________________
```

Figure 4.6: Summary of neural network with dropout layers

The second network with 2 dropout layers and 2 dense layers was proposed to test any potential measurable improvements in performance when reducing the number of layers and introducing dropout.

4.4 Model Summary

A summarized diagram of the model is shown in Figure 4.7. The entirety of the model consists of the following:

- Extract header and platform agnostic features from the PE file.
- Use the hashing trick [WDA$^+$09] to summarize header features.
- Flatten the features into a one dimensional input vector of size 2351.
- Normalize the features using statistical or z-score normalization.
- Feed the input vector through a densely connected deep neural network to obtain a binary output.

The final output of this model is either a 0 or a 1. Where 0 indicates that the PE file being tested is benign, and 1 indicates that the file is malicious.

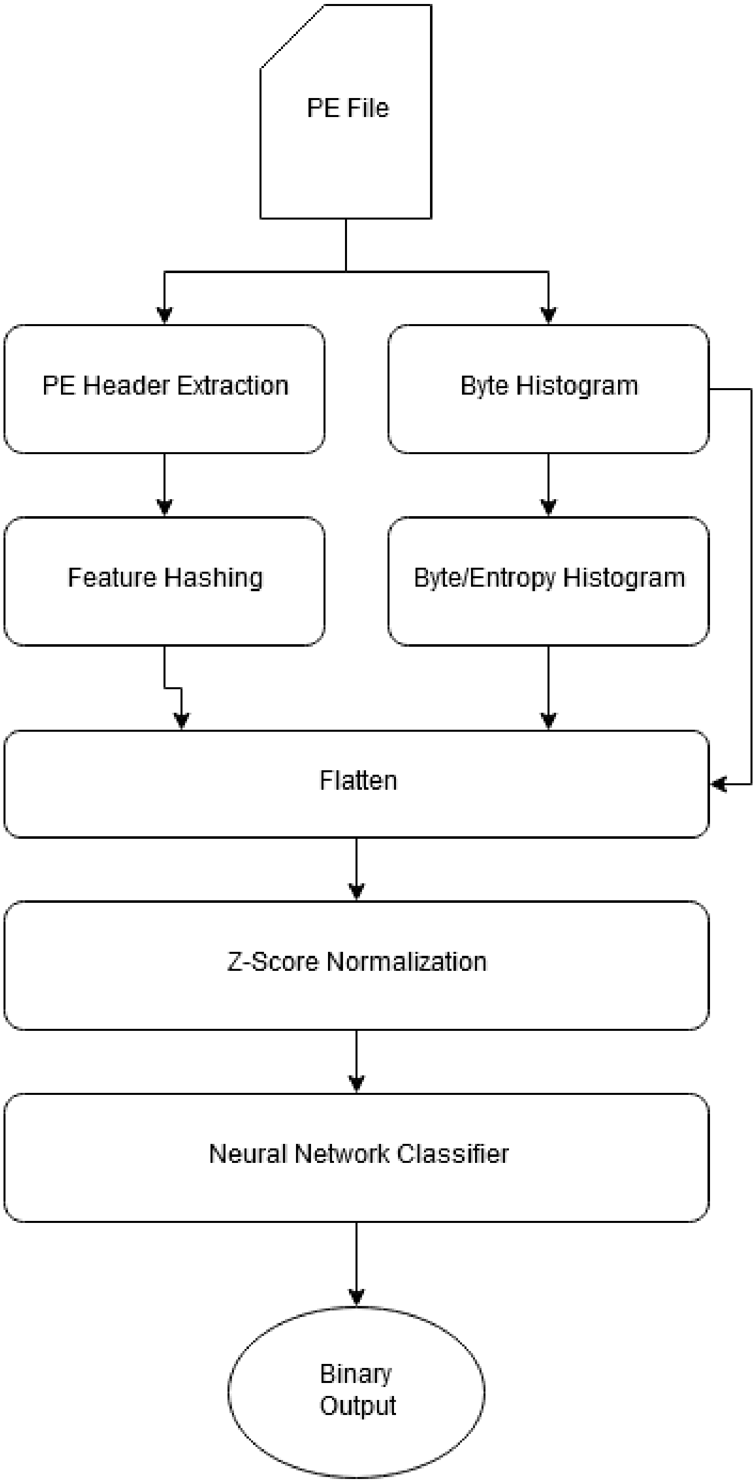


Figure 4.7: Flow diagram of the model

Chapter 5

Experiments and Results

Here, we cover all experiments performed on our models and the steps taken to implement a working classifier. We discuss the performance of our model and compare it to other models.

5.1 Experimental Setup

Our model was trained on a Dell Precision Tower with an Intel Xeon E3 processor, Nvidia GeForce GTX 1080 Ti graphics card and 64GB of RAM. We implemented it in Python with the following libraries installed:

- TensorFlow [ABC$^+$16]
- Keras [C$^+$15]
- NumPy [VDWCV11]
- scikit-learn [PVG$^+$11]
- LIEF [Tho17]
- Pandas [M$^+$10]
- MatPlotLib [Hun07]

There are also packages and libraries which the above libraries depend on, but are typically installed automatically as part of the installation process. We use Anaconda Python [Ana16] with Python 3.6.7 for all experiments.

5.2 Metrics for Model Testing

Before testing the model, it was important to identify the metrics to be used for this purpose. In our case, we are testing the accuracy and diagnostic ability of our model. For this purpose, we derive the receiver output characteristic (ROC) curve and find the area under curve (AUC). 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 [Faw06, Met78]. We also derive the confusion matrix of the neural network based models and the decision tree based model to make it easy to spot cases of misclassification which are not visible clearly from the ROC curve.

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:

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

$$FPR = \frac{FP}{N} = \frac{FP}{FP + TN}$$

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, N is the total negative samples present in the test set, and TN is true negatives. We use the `roc_curve`, `auc`, and modules from `scikit-learn.metrics` to obtain the ROC curve, and the `confusion_matrix` module from the same library to obtain the confusion matrix. All data is plotted using MatPlotLib.

We used 200K test samples provided in the EMBER dataset [AR18] to test our model and then compared our results to a decision tree based model created by the authors of EMBER using LightGBM [KMF$^+$17].

5.3 Test Results

Classifier Type	AUC	TPR @ FPR = 0.01
Neural Network (Sigmoid)	0.998	0.981
Neural Network with Dropout (Sigmoid)	0.998	0.978
Neural Network (ReLU)	0.997	0.982
Neural Network with Dropout (ReLU)	0.997	0.989
Decision Tree using LightGBM	0.999	0.982

Table 5.1: Summary of results for neural networks based classifiers, and for decision tree based classifier

We tested our model using 4 different neural network based classifiers, and then we tested it using a decision tree based classifier. A summary of the results can be found in table 5.1.

As is evident from table 5.1 and from the ROC curves in Figures 5.1, 5.3, 5.5, and 5.7, the AUC of neural networks using the ReLU activation function is slightly lower than that of the ones using the sigmoid activation function. Moreover, although the AUC of the decision tree classifier is the highest, the true positive rate of this model is same or lower when capped at 1% false positive rate as compared to the ReLU based neural networks. The confusion matrices for the models in Figures 5.2, 5.4, 5.6, 5.8, 5.11 show a better visualization of the classification performance of each model.

Since ReLU appears to obtain the best results, we compare the ROC curve of the ReLU based neural network, and the decision tree classifier in Figures 5.9 and 5.10.

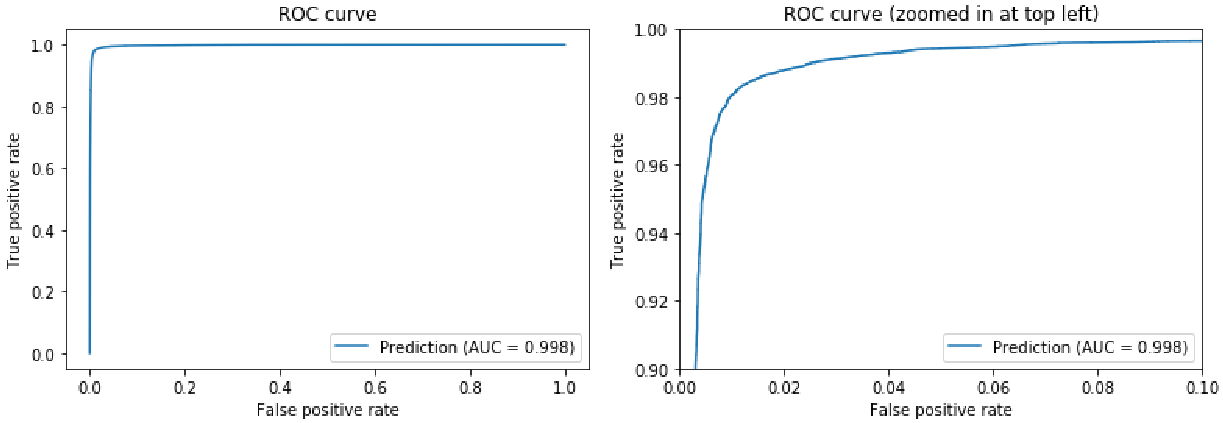


Figure 5.1: ROC Curve of model using Neural Network (Sigmoid)

5.4 Real World Testing

The final step in testing whether a model is effective or not is to attempt to test it in real world scenarios. We tested the best performing model of our proposed models (Neural Network with Dropout (RelU)) against the decision tree model to check how well they perform in detecting actual malicious PE files. The results are summarized in Table 5.2.

We used a sample set of 997 samples from VirusShare.com [Rob11] for testing. Unfortunately, the samples on VirusShare.com are accessible by registration only, hence the sample set itself cannot be shared publicly. The file name of the sample set is: `VirusShare_x86-64_WinEXE_20130711.zip`

The results obtained from the neural network based model are as expected based on the results we saw before. Of the 997 samples tested, 994 were correctly classified. The decision tree based

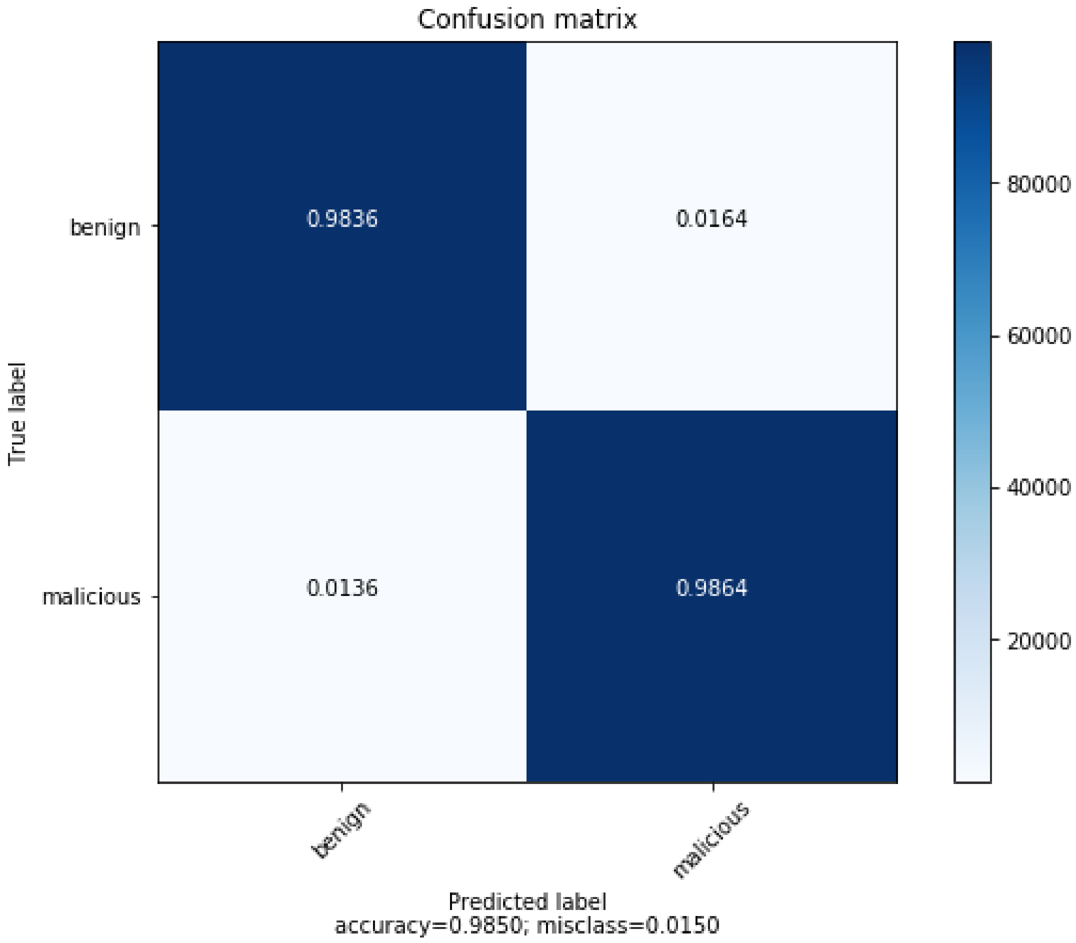


Figure 5.2: Confusion Matrix of model using Neural Network (Sigmoid)

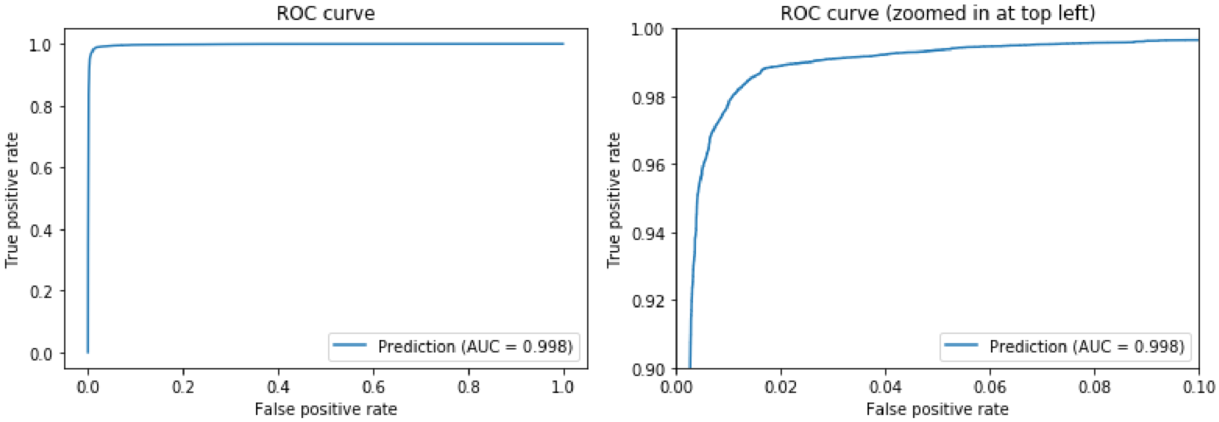


Figure 5.3: ROC Curve of model using Neural Network with Dropout (Sigmoid)

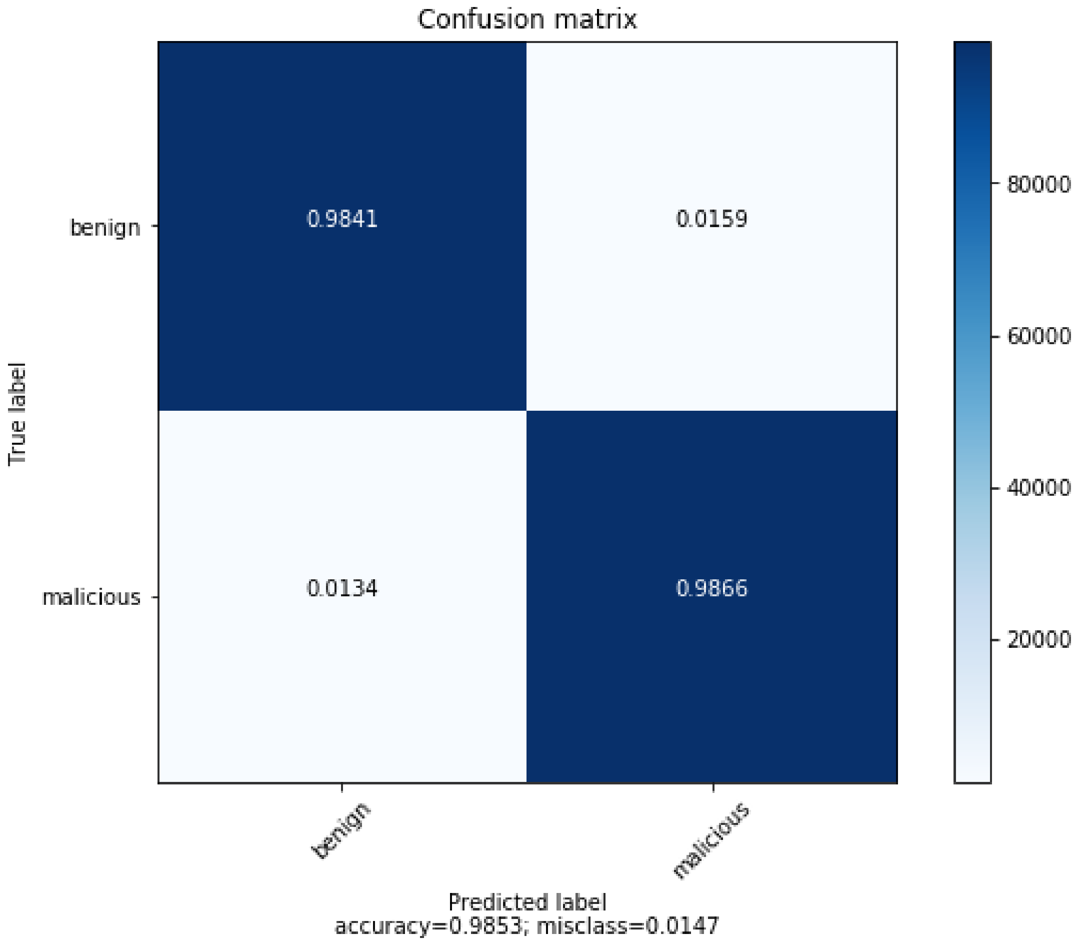


Figure 5.4: Confusion Matrix of model using Neural Network with Dropout (Sigmoid)

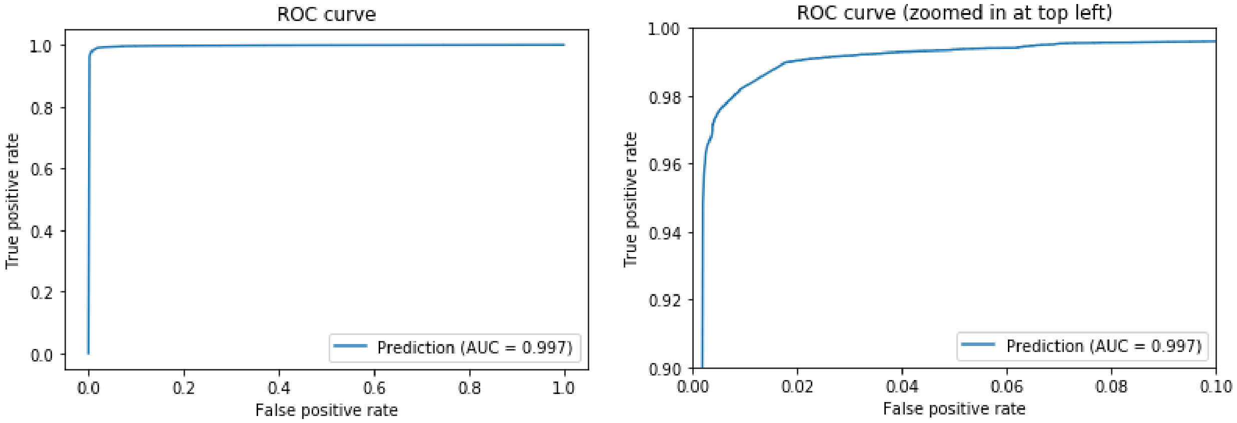


Figure 5.5: ROC Curve of model using Neural Network (ReLU)

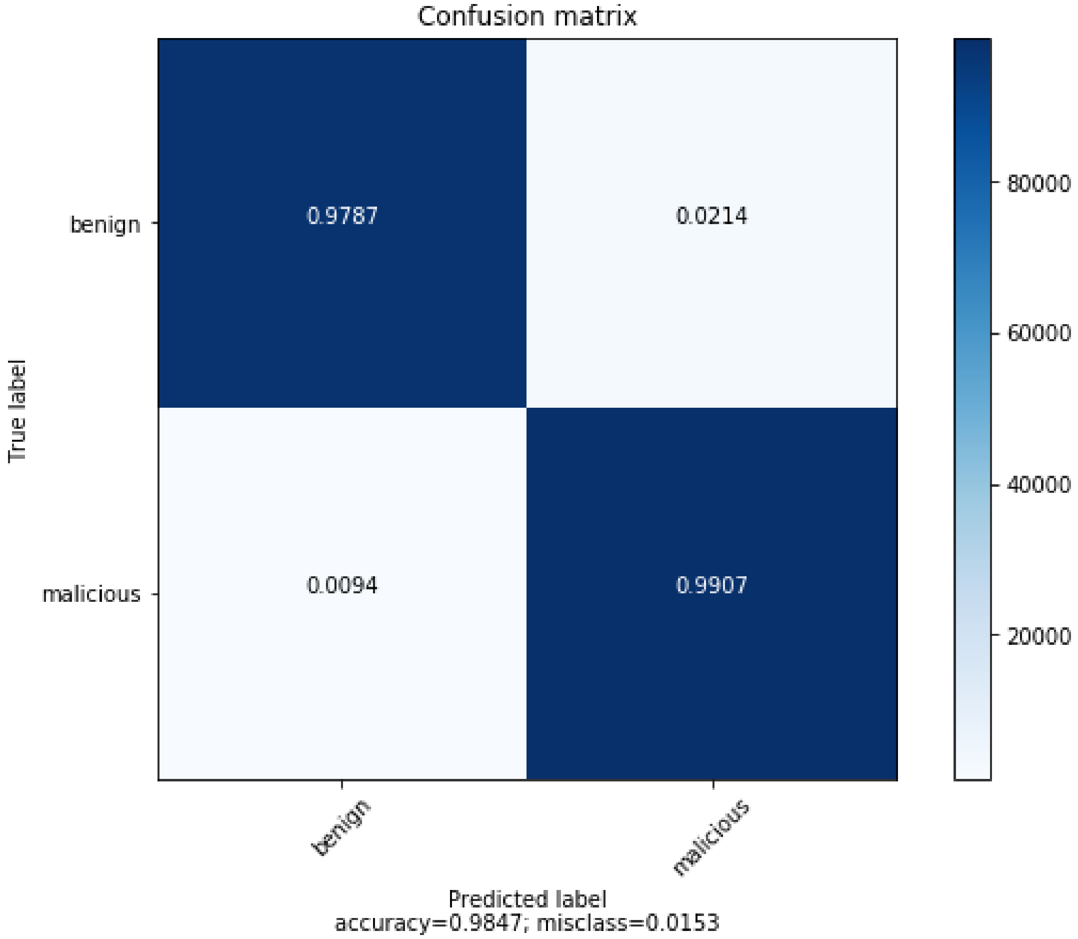


Figure 5.6: Confusion Matrix of model using Neural Network (ReLU)

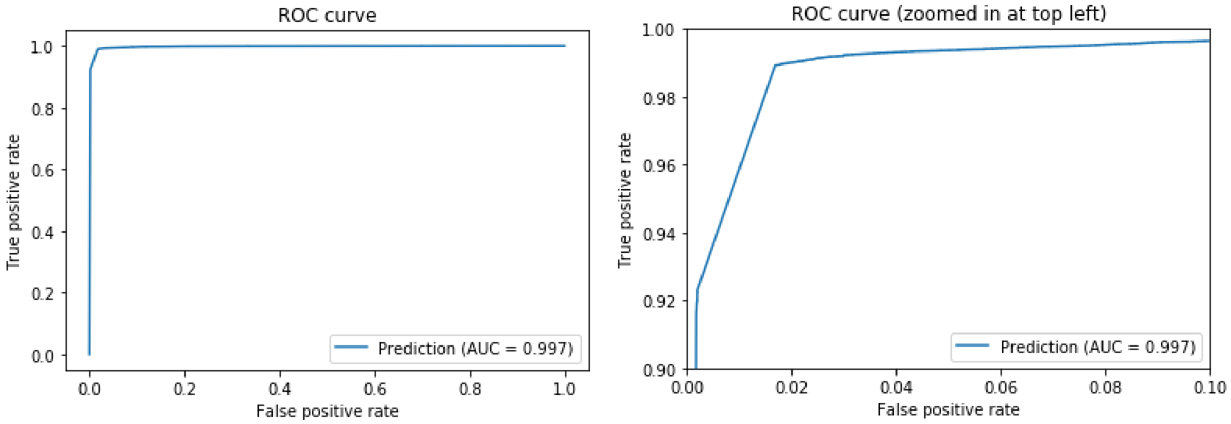


Figure 5.7: ROC Curve of model using Neural Network with Dropout (ReLU)

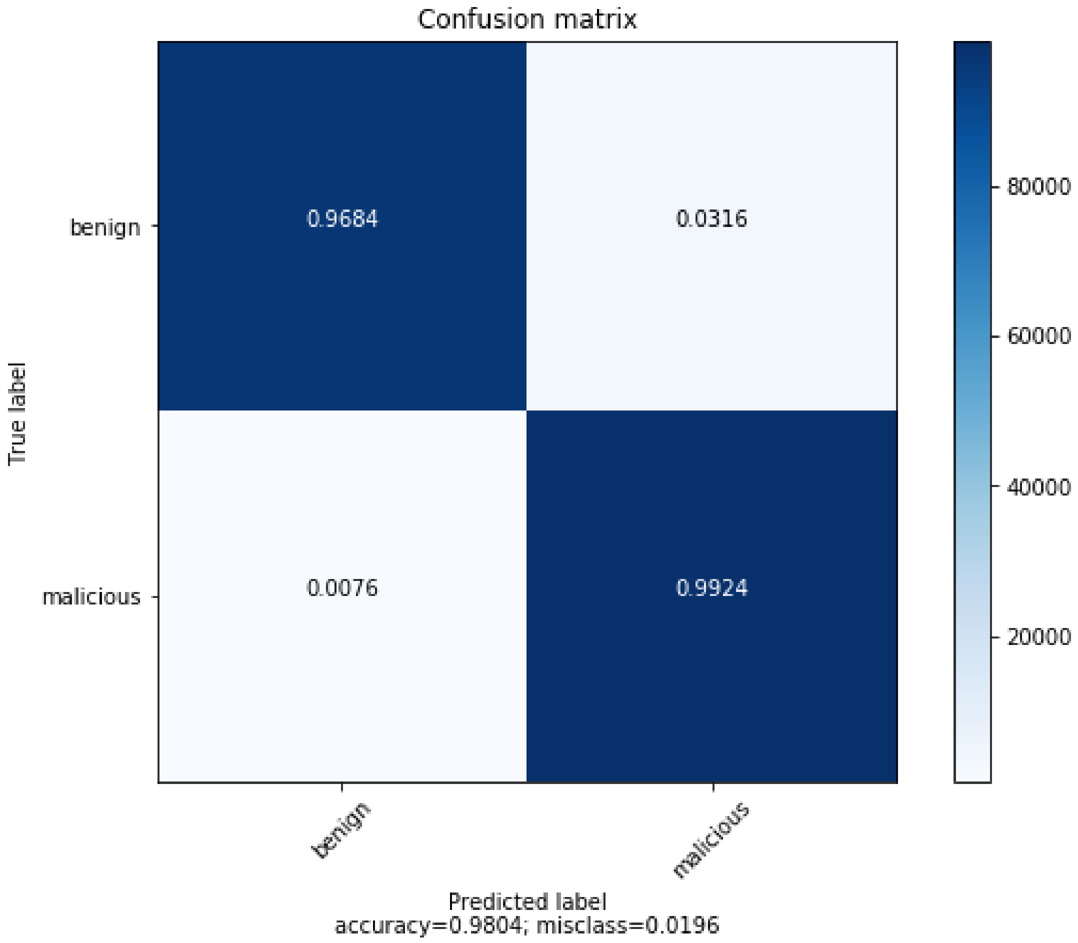


Figure 5.8: Confusion Matrix of model using Neural Network with Dropout (ReLU)

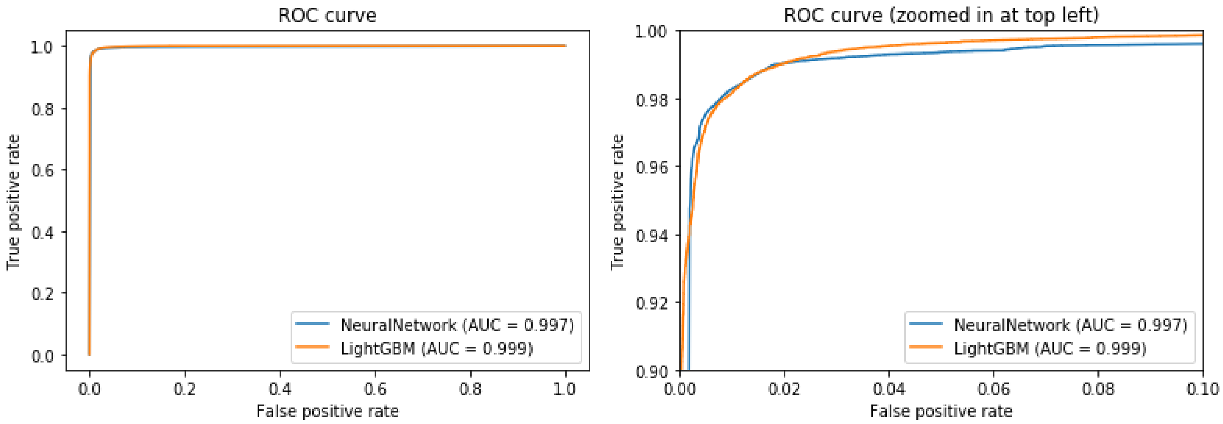


Figure 5.9: ROC Curves of model using Neural Network (ReLU) and using Decision Tree

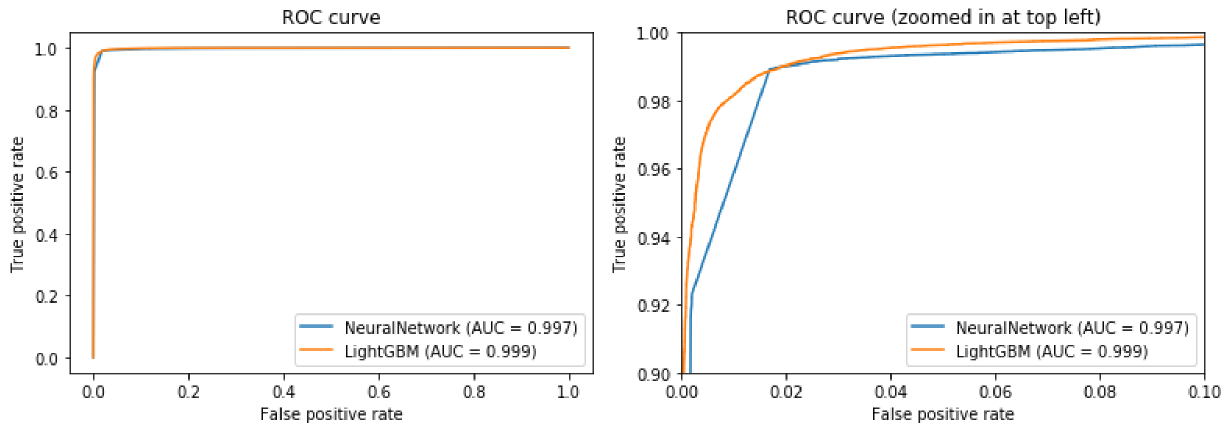


Figure 5.10: ROC Curves of model using Neural Network with Dropout (ReLU) and using Decision Tree

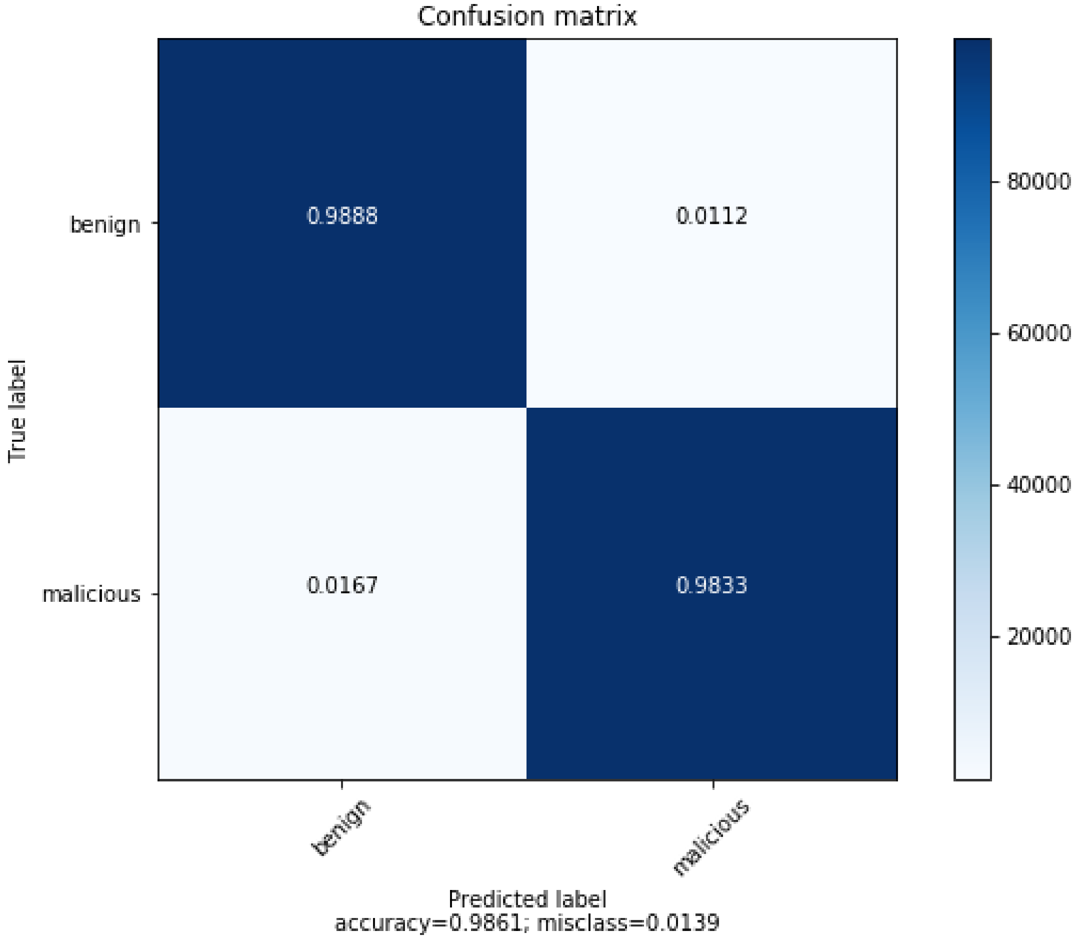


Figure 5.11: Confusion Matrix of model using Decision Tree

Model Classifier	Execution Time (seconds)	Accuracy
Neural Network with Dropout (ReLU)	128.2	0.997
Decision Tree	133.6	0.117

Table 5.2: Results from real world testing

model however, was only able to classify 117 of the 997 samples. It is difficult to determine the cause of this result without further testing.

5.5 Source Code Availability

All source code used for creating the model, experimentation, and testing is available to download on https://github.com/preppie22/malware-classifier. The dataset used for training and testing the models is available at https://github.com/endgameinc/ember. Please refer to the guide published on their page to reproduce the result of the LightGBM model discussed above.

Chapter 6

Conclusion and Future Work

In our research, we demonstrate that the use of deep neural networks for static malware detection is viable and has potential for further improvement. Our experiments show that even in situations involving structured data, the use of neural networks can still be more efficient compared to decision trees. We established a method of file vectorization that can effectively summarize large files for classification. The importance of availability of a large dataset in such domains cannot be overlooked. This research shows that static malware analysis can be an effective tool in malware classification in spite of the existence and established detection rates of dynamic malware analysis

Further research in this area will be required the establish how efficient neural networks can be as classifiers for structured data as compared to decision tree models. Real world testing has shown that there are still gaps in this area that have not been explored and will require further testing for practical implementation.

Bibliography

[ABC$^+$16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[AMS09] Srilatha Attaluri, Scott McGhee, and Mark Stamp. Profile hidden markov models and metamorphic virus detection. *Journal in computer virology*, 5(2):151–169, 2009.

[Ana16] Anaconda. Anaconda software distribution version 2-2.4.0, November 2016.

[AR18] Hyrum S Anderson and Phil Roth. Ember: an open dataset for training static pe malware machine learning models. *arXiv preprint arXiv:1804.04637*, 2018.

[Bag17] Naman Bagga. Measuring the effectiveness of generic malware models. Master's thesis, San Jose State University, 2017.

[BB01] Michele Banko and Eric Brill. Scaling to very very large corpora for natural language disambiguation. In *Proceedings of the 39th annual meeting on association for computational linguistics*, pages 26–33. Association for Computational Linguistics, 2001.

[BHL$^+$08] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. Automatically identifying trigger-based behavior in malware. In *Botnet Detection*, pages 65–88. Springer, 2008.

[Bil07] Daniel Bilar. Opcodes as predictor for malware. *International Journal of Electronic Security and Digital Forensics*, 1(2):156–168, 2007.

[BKM07] Guillaume Bonfante, Matthieu Kaczmarek, and Jean-Yves Marion. Control flow graphs as malware signatures. In *International workshop on the Theory of Computer Viruses*, 2007.

[C$^+$15] François Chollet et al. Keras. https://keras.io, 2015.

[Cen19] Windows Dev Center. Pe format - windows applications, Mar 2019. https://docs.microsoft.com/en-us/windows/desktop/debug/pe-format.

[CJ06] Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. Technical report, WISCONSIN UNIV-MADISON DEPT OF COMPUTER SCIENCES, 2006.

[CNM06] Rich Caruana and Alexandru Niculescu-Mizil. An empirical comparison of supervised learning algorithms. In *Proceedings of the 23rd international conference on Machine learning*, pages 161–168. ACM, 2006.

[Com16] Wikimedia Commons. Portable executable 32 bit structure in svg fixed, 2016. https://commons.wikimedia.org/wiki/File:Portable_Executable_32_bit_Structure_in_SVG_fixed.svg.

[DPJ15] Hamid Divandari, Bassir Pechaz, and Majid Vafaie Jahan. Malware detection using markov blanket based on opcode sequences. In *2015 International Congress on Technology, Communication and Knowledge (ICTCK)*, pages 564–569. IEEE, 2015.

[EMO12] Ammar AE Elhadi, Mohd A Maarof, and Ahmed H Osman. Malware detection based on hybrid signature behaviour application programming interface call graph. *American Journal of Applied Sciences*, 9(3):283, 2012.

[ESKK12] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM computing surveys (CSUR)*, 44(2):6, 2012.

[Faw06] Tom Fawcett. An introduction to roc analysis. *Pattern recognition letters*, 27(8):861–874, 2006.

[HRZZ09] Trevor Hastie, Saharon Rosset, Ji Zhu, and Hui Zou. Multi-class adaboost. *Statistics and its Interface*, 2(3):349–360, 2009.

[HS16] Wenyi Huang and Jack W Stokes. Mtnet: a multi-task neural network for dynamic malware classification. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 399–418. Springer, 2016.

[HSKS03] Katherine Heller, Krysta Svore, Angelos D Keromytis, and Salvatore Stolfo. One class support vector machines for detecting anomalous windows registry accesses. In *ICDM Workshop on Data Mining for Computer Security*, 2003.

[Hun07] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[JS11] T Jayalakshmi and A Santhakumaran. Statistical normalization and back propagation for classification. *International Journal of Computer Theory and Engineering*, 3(1):1793–8201, 2011.

[Kat93] Randy Kath. The portable executable file format from top to bottom. *MSDN Library, Microsoft Corporation*, 1993.

[KB14] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[Ker16] Dilshan Keragala. Detecting malware and sandbox evasion techniques. *SANS Institute InfoSec Reading Room*, 16, 2016.

[KM04] Jeremy Z Kolter and Marcus A Maloof. Learning to detect malicious executables in the wild. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 470–478. ACM, 2004.

[KMF+17] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems*, pages 3146–3154, 2017.

[KZWE16] Bojan Kolosnjaji, Apostolis Zarras, George Webster, and Claudia Eckert. Deep learning for classification of malware system call sequences. In *Australasian Joint Conference on Artificial Intelligence*, pages 137–149. Springer, 2016.

[M+10] Wes McKinney et al. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, volume 445, pages 51–56. Austin, TX, 2010.

[Met78] Charles E Metz. Basic principles of roc analysis. In *Seminars in nuclear medicine*, volume 8, pages 283–298. Elsevier, 1978.

[MKK07] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 421–430. IEEE, 2007.

[MRS10] Christopher Manning, Prabhakar Raghavan, and Hinrich Schütze. Introduction to information retrieval. *Natural Language Engineering*, 16(1):100–103, 2010.

[NBG08] John Nickolls, Ian Buck, and Michael Garland. Scalable parallel programming. In *2008 IEEE Hot Chips 20 Symposium (HCS)*, pages 40–53. IEEE, 2008.

[OBJ09] Jon Oberheide, Michael Bailey, and Farnam Jahanian. Polypack: an automated online packing service for optimal antivirus evasion. In *Proceedings of the 3rd USENIX conference on Offensive technologies*, pages 9–9. USENIX Association, 2009.

[Oli07] Travis E Oliphant. Python for scientific computing. *Computing in Science & Engineering*, 9(3):10–20, 2007.

[OSM11] Philip OKane, Sakir Sezer, and Kieran McLaughlin. Obfuscation: The hidden malware. *IEEE Security & Privacy*, 9(5):41–47, 2011.

[PCJD07] Mila Dalla Preda, Mihai Christodorescu, Somesh Jha, and Saumya Debray. A semantics-based approach to malware detection. *ACM SIGPLAN Notices*, 42(1):377–388, 2007.

[PSS+15] Razvan Pascanu, Jack W Stokes, Hermineh Sanossian, Mady Marinescu, and Anil Thomas. Malware classification with recurrent networks. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1916–1920. IEEE, 2015.

[PVG+11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[Qui93] Ross Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[R+12] Karthik Raman et al. Selecting features to classify malware. *InfoSec Southwest*, 2012, 2012.

[Rob11] J-Michael Roberts. Virus share.(2011). *URL https://virusshare. com*, 2011.

[RRF+18] Royi Ronen, Marian Radu, Corina Feuerstein, Elad Yom-Tov, and Mansour Ahmadi. Microsoft malware classification challenge. *arXiv preprint arXiv:1802.10135*, 2018.

[RYZ+05] Byron P Roe, Hai-Jun Yang, Ji Zhu, Yong Liu, Ion Stancu, and Gordon McGregor. Boosted decision trees as an alternative to artificial neural networks for particle identification. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 543(2-3):577–584, 2005.

[SB15] Joshua Saxe and Konstantin Berlin. Deep neural network based malware detection using two dimensional binary program features. In *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 11–20. IEEE, 2015.

[Sch03] Robert E Schapire. The boosting approach to machine learning: An overview. In *Nonlinear estimation and classification*, pages 149–171. Springer, 2003.

[SDB+13] Igor Santos, Jaime Devesa, Felix Brezo, Javier Nieves, and Pablo Garcia Bringas. Opem: A static-dynamic approach for machine-learning-based malware detection. In *International Joint Conference CISIS'12-ICEUTE 12-SOCO 12 Special Sessions*, pages 271–280. Springer, 2013.

[SH12] M. Sikorski and A. Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software.* No Starch Press, 2012.

[Tho17] Romain Thomas. Lief - library to instrument executable formats. https://lief.quarkslab.com/, April 2017.

[VDWCV11] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22, 2011.

[WDA+09] Kilian Weinberger, Anirban Dasgupta, Josh Attenberg, John Langford, and Alex Smola. Feature hashing for large scale multitask learning. *arXiv preprint arXiv:0902.2206*, 2009.

[WH14] Wen-Chieh Wu and Shih-Hao Hung. Droiddolphin: A dynamic android malware detection framework using big data and machine learning. In *Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems*, RACS '14, pages 247–252, New York, NY, USA, 2014. ACM.

Curriculum Vitae

Graduate College
University of Nevada, Las Vegas

Piyush Aniruddha Puranik
piyushpuranik@gmail.com

Degrees:

Bachelor of Engineering - Computer (2015)
Savitribai Phule Pune University, Pune, India

Thesis Title: Static Malware Detection using Deep Neural Networks on Portable Executables

Thesis Examination Committee:

Chairperson, Dr. Justin Zhan, Ph.D.
Committee Member, Dr. Hal Berghel, Ph.D.
Committee Member, Dr. Kazem Taghva, Ph.D.
Graduate Faculty Representative, Dr. Tiberio Garza, Ph.D.