

Android malware detection using Machine Learning methods

Bc. Jan Zdražil

Master's thesis
2023



Tomas Bata University in Zlín
Faculty of Applied Informatics

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2022/2023

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	Bc. Jan Zdražil
Osobní číslo:	A21186
Studijní program:	N0613A140022 Informační technologie
Specializace:	Kybernetická bezpečnost
Forma studia:	Prezenční
Téma práce:	Detekce malwaru běžícího pod operačním systémem Android s využitím metod strojového učení
Téma práce anglicky:	Android Malware Detection Using Machine Learning Methods

Zásady pro vypracování

1. Vypracujte literární rešerši, která mapuje současný stav řešené problematiky.
2. Provedte extrakci charakteristik aplikací běžících pod Android OS, které mají detekční potenciál.
3. Navrhněte a implementujte neuronové sítě, které budou vhodné pro detekci škodlivých aplikací.
4. Využijte nových metod strojového učení pro získání experimentálních výsledků.
5. Implementujte model složený z metod strojového učení, které vykazovaly nejlepší detekční výsledky.

Forma zpracování diplomové práce: **tištěná/elektronická**
Jazyk zpracování: **Angličtina**

Seznam doporučené literatury:

1. MOHRI, Mehryar, Afshin ROSTAMIZADEH a Ameet TALWALKAR. Foundations of Machine Learning. MIT Press: MIT Press, 2012. ISBN 9780262018258.
2. GALUSHKIN, Alexander. Neural Networks Theory. Springer-Verlag Berlin a Heidelberg GmbH & Co., 2010. ISBN 3642080065.
3. GU, Jiuxiang, Zhenhua WANG a Jason KUEN. Recent Advances in Convolutional Neural Networks. Sciencedirect [online]. 2017, 38 [cit. 2022-09-24]. dostupné z: <https://www.sciencedirect.com/science/article/abs/pii/S0031320317304120>.
4. YAN, Jinpei, Yong QI a Qifan RAO. Detecting Malware with an Ensemble Method Based on Deep Neural Network. Hindawi Security and Communication Networks [online]. 2017, 16 [cit. 2022-09-24]. dostupné z: <https://www.hindawi.com/journals/scn/2018/7247095>.
5. SAXE, Joshua and Konstantin BERLIN. Deep Neural Network Based Malware Detection Using Two Dimensional Binary Program Features. IEEE [online]. 2015, 20 [cit. 2022-09-24]. dostupné z: <https://ieeexplore.ieee.org/document/7413680>.
6. VASWANI, Ashish, Noam SHAZEER, Niki PARMAR a Jakob USZKOREIT. Attention Is All You Need. NIPS [online]. 2017, 11 [cit. 2022-09-24]. dostupné z: <https://papers.nips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-abstract.html>.

Vedoucí diplomové práce: **Ing. Milan Oulehla, Ph.D.**
Ústav informatiky a umělé inteligence

Datum zadání diplomové práce: **2. prosince 2022**
Termín odevzdání diplomové práce: **26. května 2023**



doc. Ing. Jiří Vojtěšek, Ph.D. v.r.
děkan

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Ve Zlíně dne 7. prosince 2022

I hereby declare that:

- I understand that by submitting my Diploma thesis, I agree to the publication of my work according to Law No. 111/1998, Coll., On Universities and on changes and amendments to other acts (e.g. the Universities Act), as amended by subsequent legislation, without regard to the results of the defence of the thesis.
- I understand that my Diploma Thesis will be stored electronically in the university information system and be made available for on-site inspection, and that a copy of the Diploma/Thesis will be stored in the Reference Library of the Faculty of Applied Informatics, Tomas Bata University in Zlin, and that a copy shall be deposited with my Supervisor.
- I am aware of the fact that my Diploma Thesis is fully covered by Act No. 121/2000 Coll. On Copyright, and Rights Related to Copyright, as amended by some other laws (e.g. the Copyright Act), as amended by subsequent legislation; and especially, by §35, Para. 3.
- I understand that, according to §60, Para. 1 of the Copyright Act, TBU in Zlin has the right to conclude licensing agreements relating to the use of scholastic work within the full extent of §12, Para. 4, of the Copyright Act.
- I understand that, according to §60, Para. 2, and Para. 3, of the Copyright Act, I may use my work - Diploma Thesis, or grant a license for its use, only if permitted by the licensing agreement concluded between myself and Tomas Bata University in Zlin with a view to the fact that Tomas Bata University in Zlin must be compensated for any reasonable contribution to covering such expenses/costs as invested by them in the creation of the thesis (up until the full actual amount) shall also be a subject of this licensing agreement.
- I understand that, should the elaboration of the Diploma Thesis include the use of software provided by Tomas Bata University in Zlin or other such entities strictly for study and research purposes (i.e. only for non-commercial use), the results of my Diploma Thesis cannot be used for commercial purposes.
- I understand that, if the output of my Diploma Thesis is any software product(s), this/these shall equally be considered as part of the thesis, as well as any source codes, or files from which the project is composed. Not submitting any part of this/these component(s) may be a reason for the non-defence of my thesis.

I herewith declare that:

- I have worked on my thesis alone and duly cited any literature I have used. In the case of the publication of the results of my thesis, I shall be listed as co-author.
- That the submitted version of the thesis and its electronic version uploaded to IS/STAG are both identical.

In Zlin; dated:

Jan Zdražil v.r
Student's Signature

ABSTRACT

The research explores the use of machine learning algorithms to detect malicious applications running on the Android operating system. The focus is on the identification of features from Android applications that exhibit detection potential and the subsequent creation of machine learning models. This research contributes to address the growing concern of Android malware and offers a practical solution to protect mobile devices from this threat. Using state-of-the-art machine learning methods, the output in the form of a trained model will be used as a tool in the PT-LAB penetration lab in follow-up research.

Keywords: Machine learning, Malware applications, Android operating system, Artificial neural networks, Transformers

ABSTRAKT

Výzkum se zabývá využitím algoritmů strojového učení k detekci škodlivých aplikací běžících pod operačním systémem Android. Důraz je kladen na identifikaci charakteristik mobilního malwaru, které vykazují detekční potenciál a následnou tvorbou modelů strojového učení. Tento výzkum přispívá k řešení rostoucí obavy z androidího malwaru a nabízí praktické řešení pro ochranu mobilních zařízení před touto hrozbou. Díky využití nejmodernějších metod strojového učení bude výstup v podobě naučeného modelu využit jako nástroj v penetrační laboratoři PT-LAB v navazujících výzkumech.

Klíčová slova: Strojové učení, Škodlivé aplikace, Operační systém Android, Umělé neuronové sítě, Transformátory

My gratitude goes to the members of the PTLAB for providing the data without which this study would not have been possible. I would also like to express my thanks to my supervisor Ing. Milan Oulehla, Ph.D., who has guided this thesis in an exemplary manner. Finally, I would like to thank my whole family, who have always been and still are my greatest support not only in my studies but also in my life.

I hereby declare that the print version of my Master's thesis and the electronic version of my thesis deposited in the IS/STAG system are identical.

CONTENTS

INTRODUCTION	10
I THEORY.....	11
1 RESEARCH INTO THE PROBLEM OF DETECTING MALICIOUS APPLICATIONS ON THE ANDROID PLATFORM USING MACHINE LEARNING METHODS	12
1.1 Q-1: WHAT STATIC ANALYSIS TECHNIQUES ARE USED TO DETECT ANDROID MALWARE?	13
1.2 Q-2: HOW SHOULD A STUDY ON ANDROID MALWARE DETECTION PROCEED?	14
1.3 Q3: WHAT AUXILIARY TOOLS CAN BE USED FOR STATIC ANALYSIS OF ANDROID MALWARE DETECTION?	14
1.4 Q4: WHAT MACHINE LEARNING MODELS ARE BEING APPLIED TO THE PROBLEM OF ANDROID MALWARE DETECTION?.....	15
2 ANDROID OPERATING SYSTEM	17
2.1 ANDROID OPERATING SYSTEM ARCHITECTURE	18
2.1.1 Linux Kernel	19
2.1.2 Hardware Abstraction Layer	19
2.1.3 Android Runtime.....	20
2.1.4 Native C/C++ Libraries.....	21
2.1.5 Java API Framework.....	22
2.1.6 System Apps.....	22
2.2 ANDROID APPS.....	22
2.2.1 Activities	23
2.2.2 Services	23
2.2.3 Broadcast receivers	24
2.2.4 Content providers	24
2.3 ARCHITECTURE OF APK.....	25
2.3.1 AndroidManifest.xml	25
2.3.2 Meta-INF	26
2.3.3 Resources.arsc	26
2.3.4 Classes.dex	26
2.3.5 Res.....	27
2.3.6 Assets	27
2.3.7 Lib	27
3 ARTIFICIAL NEURAL NETWORKS.....	28
3.1 ARCHITECTURE OF NEURAL NETWORKS	29
3.1.1 Neurons	29
3.1.2 Layer types	30
3.2 TRAINING OF NEURAL NETWORKS	31
3.3 ACTIVATION FUNCTIONS	32
3.4 CONVOLUTION NEURAL NETWORKS	34

4	TRANSFORMERS	37
4.1	MODEL ARCHITECTURE	37
4.1.1	Encoder and Decoder	38
4.1.2	Attention.....	39
4.2	VISION TRANSFORMER (ViT)	40
5	HYPER-PARAMETER OPTIMIZATION	42
5.1	METHODS OF HYPER-OPTIMIZATION	42
5.2	BAYESIAN OPTIMIZATION	43
5.3	TREE-STRUCTURED PARZEN ESTIMATOR	44
II	ANALYSIS.....	45
6	DATA GATHERING.....	46
7	FEATURE EXTRACTION FROM AN ANDROID APPLICATION WITH DETECTION POTENTIAL	50
7.1	APK TO GREYSCALE IMAGE APPROACH.....	50
7.2	DEX TO GREYSCALE IMAGE APPROACH	52
7.3	DEX TO RGB IMAGE APPROACH.....	54
7.4	EXTRACTION OF ANDROIDMANIFEST.XML PROPERTIES	57
8	SELECTION OF EXTRACTED FEATURES FOR INPUT TO MACHINE LEARNING METHODS.	60
9	RESEARCHING AND UTILIZING NEURAL NETWORKS ON THE PROBLEM OF MALICIOUS APPLICATION DETECTION.....	64
9.1	ARCHITECTURE OF THE PROPOSED MODEL	65
9.2	KEY FEATURES OF MD-NNM	67
9.3	PREVIEW OF MD-NNM TRAINING.....	68
10	UTILIZATION AND EXPLORATION OF NOVEL MACHINE LEARNING METHOD TO ACHIEVE EXPERIMENTAL RESULTS.....	70
10.1	ARCHITECTURE OF THE EXPERIMENTAL MODEL	70
10.2	PREVIEW OF MD-ViTNNM TRAINING	72
11	OPTIMIZATION AND RESULTS OF THE OBTAINED MODELS.....	74
11.1	OPTIMIZATION PROCESS.....	75
11.2	RESULTS AND DISCUSSION.....	76
12	IMPLEMENTATION OF THE SELECTED MODEL PRODUCING THE BEST RESULTS FOR MALICIOUS APPLICATION DETECTION ON ANDROID PLATFORM.	78
12.1	PROJECT FUNCTIONALITY DEMONSTRATION.....	79
	CONCLUSION	82
	BIBLIOGRAPHY	83

LIST OF ABBREVIATIONS91
LIST OF FIGURES93
LIST OF TABLES95

INTRODUCTION

With the rapid development of the Google Play platform, there has been a noticeable increase in malicious applications targeting end user. According to statistics published by AV-Test, around 450 thousand of new instances of malicious code have been detected on a daily basis in 2022 [1]. Another alarming fact highlighting this research's importance is a Panda Security report stating that Android devices are 50 times more likely to be infected with malware relative to iOS [2]. One of the reasons for the current situation is the security policy and transparency for users on Google Play, the primary distribution platform. In addition, a significant number of mobile devices use non-paid antivirus programs (some mobile devices do not use any) which, according to a survey by Comparitech, are not able to properly detect malicious code in 53 percent [3]. Already existing methods for analyzing malicious code can be divided into three approaches. Static analysis where manual or automatic analysis of the source code and resources is used without running the application itself, dynamic analysis which is applied to a running program on a real or virtual device and hybrid analysis which combines the previous two approaches. Despite the fact that dynamic analysis is nowadays considered powerful in the hands of an experienced professional, its development and automation is very challenging both in terms of time and its dependence on human interaction [4]. One of the more popular static analysis techniques for detecting malicious code is based on signature scanning, which has a number of advantages such as the simplicity of the method and the ability to provide almost non-existent false positive rate [5]. Detection based on signature technique employs pre-defined signatures or identified patterns, which can be matched with the strings or binary pattern information found inside the targeted file. A significant drawback of this technique is the need for a signature database that must be continuously updated. Consequently, the signature-based method is unable to detect new or unknown malware [5]. In response to the shortcomings of signature-based methods and the time-consuming nature of dynamic methods, in recent years, there has been a transition to machine learning-based techniques [6]. A notable benefit of machine learning models over traditional methodologies is that when trained on sufficient data (Android applications), these systems are able to detect malicious applications that have not been included in the training but contain similar properties [6]. Therefore, this research is also directly aimed at feature extraction and selection from Android applications and their subsequent application to custom-built models for effective malicious code detection on Android operating system.

I. THEORY

1 RESEARCH INTO THE PROBLEM OF DETECTING MALICIOUS APPLICATIONS ON THE ANDROID PLATFORM USING MACHINE LEARNING METHODS

The first step towards the completion of an academic thesis is to conduct detailed research on the current state of the art, which is the main subject of this chapter. With the availability to access several databases such as *IEEE Xplore Digital Library* [7], *ScienceDirect* [8], *SpringerLink* [9] and *Google scholar* [10], it is possible to find a considerable number of scientific papers dealing with the problematics of malware detection. Since the aim of the thesis is detection of Android malware with the help of machine learning methods, this chapter will also primarily focus on the comparison of scientifically related works. Malware detection can generally be divided into static, dynamic, and hybrid analysis. Recently, there has been a considerable worldwide effort to primarily employ automated static analysis mainly due to its easier automation relative to the dynamic analysis [4]. This claim is also supported by the scientific paper "*A Systematic Literature Review of Android Malware Detection Using Static Analysis*" [11], which discusses the recent trend of Android malware detection and the methodologies used showing the most promising results. In order to achieve a meaningful outcome from this comparative study, a set of relevant questions is raised and answered in sequence. The set of selected questions is presented in Table 1.

Table 1. *Set of defined questions.*

ID	Defined questions
Q-1	What static analysis techniques are used to detect Android malware?
Q-2	How should a study on Android malware detection proceed?
Q-3	What auxiliary tools are used for static analysis of Android malware detection?
Q-4	What machine learning models are being applied to the problem of Android malware detection?

1.1 Q-1: What static analysis techniques are used to detect Android malware?

In general, the Android Application Package (APK) contains several main components: META-INF, libraries, resources, assets, classes.dex and AndroidManifest.xml, (see the subchapter *Architecture of APK*). Depending on which part of the APK the features are extracted from, the static analysis can be divided into three main categories.

1, Android characteristic-based method

This category covers the extraction of general features associated with an Android characteristic method. One of the main sources of properties related to this category are bytecode programs which are running on the virtual machine that may take the form of dex files or smali files, obtained after disassembling the dexes. In addition, there are also additional type of features located in Androidmanifest.xml such as permissions, resources, intents, and hardware components, which can be considered as feature vectors and thus can be represented for example by an image [11]. Such features can be further used to train machine learning models [11]. For example, in the article "*Machine Learning for Android Malware Detection Using Permission and API Calls*" the methodology using permissions and API calls to detect Android malware is described [12].

2, Opcode-based method

The opcodes are obtained from dex files of the APK. This method treats the opcode sequence as text and applies models based on natural language processing or statistical methods to detect whether it is malware or a clean application [11]. As an example, in the book "*Android Malware Detection using Machine Learning*", part of the process of detecting an Android application consists of extracting n-grams and then transforming them into fuzzy hashes [13].

3, Program Graph-based method

This method is based on a graph program providing an abstract structure of a given program unit. In case of Android malware detection, the program graph is extracted from APK binaries. Such a procedure captures both the syntactic and semantic components, which can be further used for subsequent prediction. The scientific paper "*Android malware familial classification and representative sample selection via frequent subgraph analysis*"

deals with extraction of the function call graphs and production of frequency subgraphs that represent the behavior of different malware families [14].

1.2 Q-2: How should a study on Android malware detection proceed?

Here, like the majority of studies using machine learning methods, it is strongly recommended to follow the data science cycle [15]. Therefore, this research can be summarized in five points. First, it is important to define and understand the problem, in this case the detection of Android malware. Next, it is necessary to gather clean and malicious applications that will represent the dataset, preferably in a one-to-one ratio. The following step is the extraction and selection of APK features that exhibit strong detection potential. Where these features will act as input to the selected models. Subsequently, it is required to select the proper machine learning method and modify its architecture to suit the problem. Finally, hyper-optimization of the parameters of the developed model should be performed to achieve the best results [16].

1.3 Q3: What auxiliary tools can be used for static analysis of Android malware detection?

As stated in the introduction, static analysis is a technique working with the source code and resources of an application. Significant advantages of this approach over dynamic analysis are: (i) Easier automation of the process due to less reliance on the human factor [4, 17]. (ii) Safety of the analysis as such since there is no need to utilize secure environments [4, 18]. For these reasons, a number of tools were developed that simplify the extraction and processing of data related to the Android application. Several popular tools together with their brief description are shown in Table 2.

Table 2. *Showcase of tools [11].*

Tool	Description
Apktool	A tool for reverse engineering in Android applications
Androguard	Reverse engineering malware/being analysis of Android applications
Aapt	A tool to compile resource files into DEX
Baksmali	Smali/baksmali is an assembler/disassembler for the dex format
Dex2jar	Tool that works with Android. dex and java .class files

Dexdump	A tool that supports decompilation of apk, dex, odex and oat files.
Dedexer	A disassembler for DEX files
Jadx	Dex to Java decompiler

1.4 Q4: What machine learning models are being applied to the problem of Android malware detection?

With the recent rapid development of machine learning, it is not surprising that the majority of modern Android malware detection techniques are using this approach. In general, the most commonly used machine learning methods in this field are Support Vector Machines, Artificial Neural Networks, Naive Bayes, Logistic Regression, k-nearest neighbors and Ensemble Learning [19]. Furthermore, it is necessary to note that according to the article "*A Systematic Literature Review of Android Malware Detection Using Static Analysis*" the combination of machine learning and features belonging to the group of Android feature-based methods is capable of achieving the finest results [11]. One of the latest approaches for Android malware detection, is transforming the entirety or just a portion (usually classes.dex) of an application into an image. A recent scientific paper that deals with this topic "*Efficient Malware Classification by Binary Sequences with One-Dimensional Convolutional Neural Networks*" shows the conversion of the application to a grayscale image, which is later processed using a convolution neural network (CNN) [20]. To get a more profound understanding of this specific class of methodologies, a collection of scientific articles is selected to provide a basis for the feature extraction and selection part of the present thesis. Since the detection of Android malware is a fairly complex problem, it is not recommended to rely on only one methodology. Therefore, in Table 3 are not summarized only scientific papers viewing the application as an image but also e.g., including features extracted from AndroiManifest.xml. Another major challenge is the proper selection of the machine learning method itself. In the field of image processing, CNN methodology is still considered the most widely employed [21]. However, with the appearance of transformers and their modifications, this method can be considered as a strong competitor to CNN. For this reason, in Table 3 are also included articles related to the selection and evolution of machine learning models and their architectures.

Table 3. *Selection of relevant scientific papers.*

The paper's title	Authors	Year of publication
Efficient Malware Classification by Binary Sequences with One-Dimensional Convolutional Neural Networks	Wei-Cheng Lin, Yi-Ren Yeh	2022
Android Malware Detection using Convolutional Neural Networks and Data Section Images	Jaemin Jung, Jongmoo Choi, Seong-je Cho	2018
Android Malware Familial Classification Based on DEX File Section Features	Yong Fang, Yangchen Gao, Fan Jing	2020
Machine Learning for Android Malware Detection Using Permission and API Calls	Naser Peiravian, Xingquan Zhu	2013
Android Malware Detection through Manifest and API Calls Tracing	Dong-Jie Wu, Ching-Hao Mao	2012
Recent advances in convolutional neural networks	Jiuxiang Gu, Zhenhua Wang	2018
Attention Is All You Need	Ashish Vaswani, Noam Shazeer	2017
AN IMAGE IS WORTH 16X16 WORDS: TRANSFORMERS FOR IMAGE RECOGNITION AT SCALE	Alexey Dosovitskiy, Lucas, Beyer	2021

At this point it should be emphasized that this is not a complete list of all the papers used in this thesis, but only the most important ones [20, 22, 23, 12, 24, 25, 26, 27]. With the knowledge obtained from the expert research, namely what kind of features to focus on, what tools to employ, how to proceed, and which machine learning methods is wise to adopt, it is now possible to proceed with the study itself.

2 ANDROID OPERATING SYSTEM

Android is the most widely used operating system across all mobile platforms worldwide. Nowadays, it is estimated that there are up to 3.3 billion android smartphone users spread across 190 countries worldwide [28]. This claim is further supported by the fact, that by the beginning of 2023, Android holds more than 70% of the total market share [29, 30]. The origins of Android as such can be traced back to the company Open Handset Alliance, whose aim was to create an accessible operating system for mobile technology. The cornerstone of the Android operating system was and still remains as a modified version of the Linux kernel and other open-source software's [31]. The defining moment for Android came in 2005 when Google LLC gained control of its development. In September 2008, the first device running under the Android operating system was officially released to the market [32]. The main reasons why Android has become so successful include its very affordable price of mobile devices relative to its competition, the free usage of the operating system for hardware manufacturers and the fact that 96.9% of applications on the Google Play Store are free [33]. Unfortunately, these benefits can also be considered the most serious disadvantages of this system, mainly due to the ease of access of criminal organizations to the ecosystem and their ability to directly distribute malicious code [34]. According to the research: *"How Did That Get In My Phone? Unwanted App Distribution on Android Devices [35]"*, the Google Play Store as well as other Android app distribution markets contain an alarming number of malicious applications, that need to be protected against by the end user [35]. In this study a vector detection ratio (VDR) is also introduced, displaying the total volume of downloaded applications relative to malicious ones. According to the study, the VDR indicator indicate that 67% of all unwanted applications come from the primary distributor Google Play Store. However, it should be noted that Google Play Store distributes up to 87% of all applications, so despite the high percentage of unwanted applications, it can still be considered the "safest" distributor in relation to alternative markets [35]. In order to get a general overview, the individual markets and their percentage distribution are shown in Table 4.

Table 4. *Summary of app distribution [35].*

Installs		
Vector	All [%]	Unwanted [%]

Playstore	87.2	67.5
Alt-market	5.7	10.4
Backup	2.0	4.8
Pkginstaller	0.7	10.5
Bloatware	0.4	6.0
PPI	0.2	0.1
Fileshare	< 0.1	< 0.1
Themes	< 0.1	< 0.1
Browser	< 0.1	< 0.1
MDM	< 0.1	< 0.1
Filemanager	< 0.1	< 0.1
IM	< 0.1	< 0.1
Other	< 0.1	< 0.1
Unclassified	3.7	< 0.1
All	100	100

2.1 Android operating system architecture

A profound understanding of the operating system's working mechanism is a prerequisite for the successful implementation of machine learning methods in malicious applications detection. Therefore, this subchapter is devoted to the description of the Android structure and its components. The Android architecture contains a diverse number of components to support any Android device [36]. The Linux kernel guarantees the basic functionality of an operating system for modern smartphones. Another essential element is the Android Runtime, providing a platform for the running of the Android application [37]. These elements and many more are essential to understand to effectively address the issue of malicious code detection for the Android platform. The fundamental division of the Android system, that will be subsequently addressed, is shown in Figure 1.

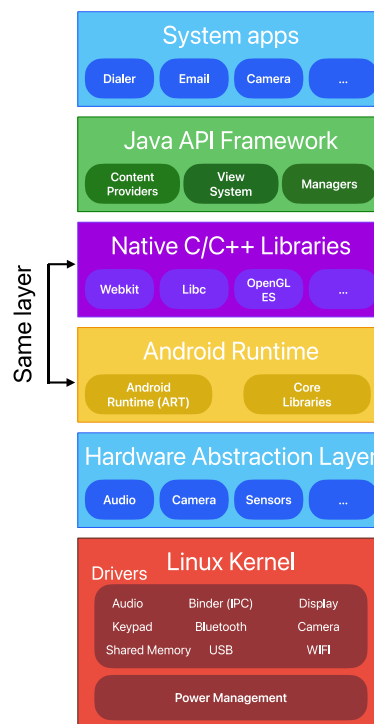


Figure 1. *Android Layer cake* [36].

In general, the model can be divided into five basic layers where each layer is based on the one underneath.

2.1.1 Linux Kernel

The Linux kernel is the bedrock of the Android architecture. It functions as the primary interface between the hardware and software of the device. It is a very robust and reliable framework due to its precise tuning. Furthermore, it should be mentioned that developing a custom kernel is very complicated and therefore a significant number of systems are based on the Linux kernel. [36]. In general, the features of the Linux kernel can be summarized as follows: (i) Security: The Linux kernel guarantees protection between the application and the system. (ii) Memory management: It deals with memory management efficiently, thus helping to provide freedom for application development. (iii) The network stack: The Linux kernel ensures communication on a certain network. (iv) Driver model: It takes care of proper functionality of applications on the targeted device [37].

2.1.2 Hardware Abstraction Layer

The Hardware Abstract Layer (HAL) grants a standard environment, exposing the capabilities of a hardware device to a higher layer known as the Java API framework [38]. The HAL is composed of several library modules, where each of them provides an interface

for a particular type of hardware component, such as a camera or Bluetooth module. When the API framework makes a call to enable access to the device hardware, the Android system retrieves the library of the module for the selected hardware component [38].

2.1.3 Android Runtime

Android runtime (ART) is the managed runtime accessed by applications and certain system services on Android. Dalvik and its successor ART have been developed specifically for the needs of the Android Operating System [39, 40]. Here it is necessary to mention that Dalvik and ART are compatible runtimes running Dex bytecode, so apps developed for Dalvik should work when running with ART. However, some techniques that work on Dalvik do not work on ART [40].

The Dalvik Virtual Machine (DVM) is a register-based virtual machine designed by Dan Bornstein [41]. It is a discontinued process on a virtual machine, which runs applications written for Android Operating System. Although Dalvik bytecode is still used as a distribution format for older versions, it has been gradually replaced by its successor [42]. In general, DVM can execute dex files and its development has undergone many improvements such as the use of just in time (JIT) technique [41]. The DVM conversion process is shown in Figure 2.

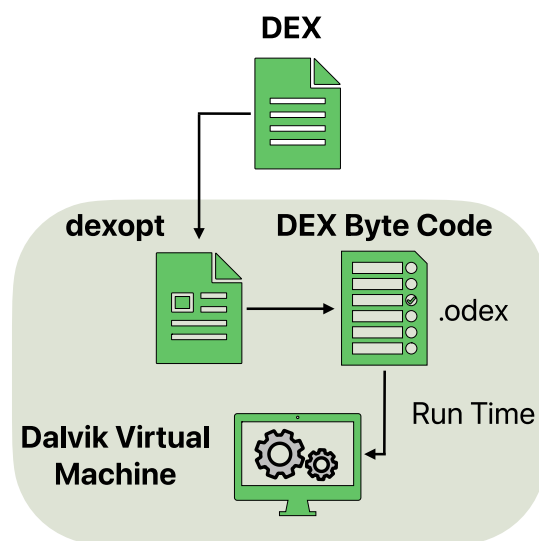


Figure 2. *Dalvik Virtual Machine* [42].

As mentioned, newer versions of Android (starting with KitKat 4.4) are increasingly transitioning to ART, which uses similar bytecodes and .dex files as DVM. [43]. The key merit of ART is to enhance the performance for end user. ART ensures this improvement by using profile-guided compilation that utilize ahead of time (AOT) and JIT techniques. [42]. Like any method, this approach has some disadvantages, such as higher storage space requirements and slightly longer installation time [43]. After installing the application, Android automatically optimizes the app data and creates a corresponding OAT file, which significantly accelerates the loading speed of the application [41, 43]. The procedure used by ART is presented in Figure 3.

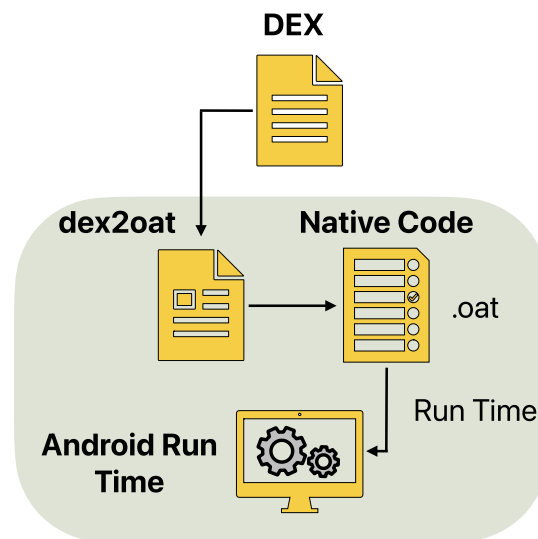


Figure 3. *Android Run Time* [38].

2.1.4 Native C/C++ Libraries

A number of core Android system components and services, including for example ART and HAL, are built from native code that relies on native libraries written in C and C++. The Android platform uses Java framework APIs to expose the functionality of some of these native libraries to applications. For example, when developing an application that requires C or C++ code, it is possible to use the Android Native Development Kit to access selected native platform libraries directly from the native code [36, 44, 45]. The Native Development Kit (NDK) is a set of toolkits, enabling the utilization of C and C++ code together with Android and providing platform libraries, that can be used to manage native activities and to open access to physical device components such as sensors [44]. In practice, NDK is often

used to gain additional performance out of a device to obtain low latency or to run computationally expensive applications (games, simulations, etc.) [44, 46].

2.1.5 Java API Framework

Using the APIs framework allows to operate the entire feature-set of the Android Operating System [46]. This framework has been developed in the Java programming language. These APIs form the basic building blocks needed to develop Android applications by providing a simplifying reuse of the core, modular system of components and services, which include the following features [36]: (i) Resource Manager, granting the access to non-code resources such as localized strings, graphics, and layout files. (ii) Notification Manager, allowing applications to show their custom alert in the status toolbar. (iii) Activity Manager, handling the application lifecycle and delivering a fundamental navigational back stack [36]. And finally, (iv) View System, which is used to develop the application UI, such as buttons, text boxes and so forth [45].

2.1.6 System Apps

System applications serve as a set of core applications, which are pre-installed into the Android system. They perform basic functions such as texting, emailing, web browsing and other essential operations [44]. Although these applications are integrated into the system, they do not possess any special status relative to other applications that the user chooses to download. Thus, a third-party application may become the user's default choice, such as a web browser or SMS messenger. Another significant features of system applications is that they provide the key capabilities for developers to create their custom applications. For example, if a developer wants to create an application that includes sending SMS, he doesn't need to create this functionality "from scratch" but can utilize the functions of system apps that are already integrated in the system [46].

2.2 Android Apps

Another important part of the Android Operating System are its components. Application components are the fundamental building blocks of any Android application. These components are loosely binded by an application manifest file (see subchapter "*Architecture of APK*" for more information about `AndroidManifest.xml`). Thus, there are four main components, namely Activities, Services, Broadcast Receivers and Content Providers, where their depiction can be seen in Figure 4.

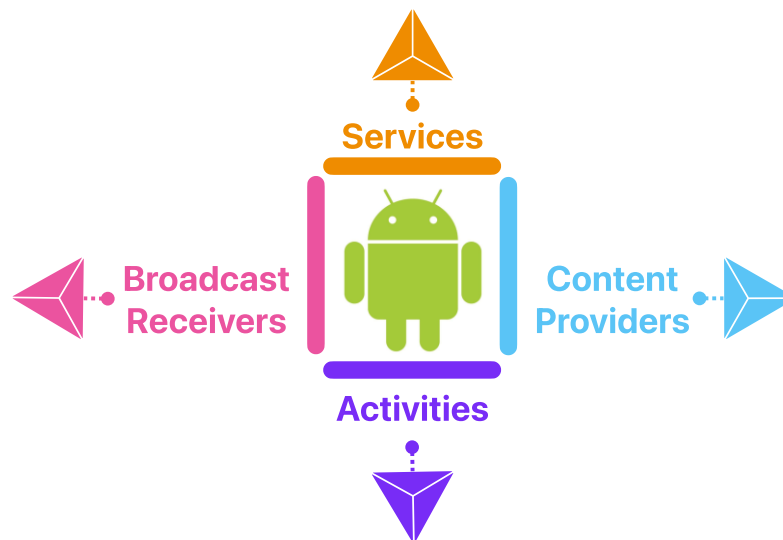


Figure 4. *Android components.*

2.2.1 Activities

An activity serves as a means for communication with the user. In general, an activity represents a single screen within a user interface. As an example, the email application may have one activity, displaying a list of incoming emails, another for reading emails, etc. If an application has more than one activity, then one of them should be marked as the activity that is presented when the application is launched [47]. Here it is necessary to mention that during the interaction with the application by the user, the activity instances undergo different states of their life cycle. The activity class provides several callbacks that inform the activity when the state has changed or that the system is creating, stopping, resuming, pausing, stopping or destroying the given activity [48].

2.2.2 Services

A service can be considered as a general-purpose means, which runs in the main thread of its hosting process. The service does not create its own thread and does not run in a separate process unless it is specified otherwise. [47]. An example of this is music running in the background, which is managed by the service even though the user is currently using another application. In general, services can be divided into two groups: bound services and started services.

Started services are being used to instruct the system to ensure their maintenance until they finish their execution. This might include, for example, data synchronization or the previously mentioned background music [49].

Bound services, as the name implies, they run because the particular application or system requires their use. It is essentially a service that provides an API to other processes [49].

It should be noted here that the official Android documentation presents yet another categorization of services into Foreground, Background and Bound and it is not clear which of these two categories is the official policy.

2.2.3 Broadcast receivers

A broadcast receiver is another important component, providing the system the ability to share an event to the application beyond the ordinary user flow. Such a process allows the app to respond to system-wide broadcast announcements [47]. One benefit of broadcast receivers is that the system is able to deliver broadcasts even when the application is not running. E. g., an application that oversees the alarm schedule can publish a notification to the user about an upcoming event *via* the broadcast receiver without running [50]. Here it should be mentioned that a considerable number of broadcasts originate from the system, such as a broadcast that the battery is running low or that the screen has been turned off. Broadcasts can also be initialized by applications to let other applications know, for instance, that certain data has been downloaded and are available for further use [51].

2.2.4 Content providers

The primary purpose of content providers is to be utilized by other applications which access the provider via the Provider Client object. A content provider presents data to external applications as single or multiple tables that are similar to tables found in a relational database [47]. In general, the work with content providers can be summarized into two scenarios: (i) Implementing code to access an existing content provider. (ii) Creating a new content provider for an application to share data with another application [52]. From the system's perspective, the content provider is a means into the application for publishing named data items that are identifiable by a URL scheme. This feature allows the system to make a number of conveniences in the process of managing the application. First, URL assignment does not require the application to continue running, so the URLs are preserved regardless of the state of the application. And second, these URLs provide

a level of security because an application can, for example, embed the URL of an image in its clipboard but keep its content provider locked down so that other applications cannot access it. [50].

2.3 Architecture of APK

To get a better understanding of the features, which are typically extracted from an Android application and further used as inputs to machine learning models, it is essential to discuss the APK architecture. An APK package must contain several key components to be considered as a valid. Namely classes.dex, res, asset, lib, AndroidManifest.xml, Meta-INF directory and resources.arcs [53]. In Figure 5 the APK file together with its key elements are depicted.

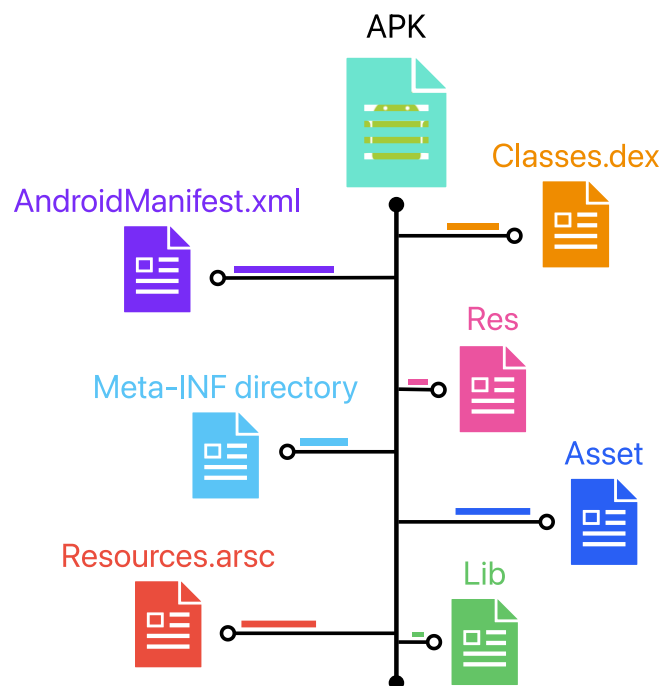


Figure 5. *Android APK structure.*

2.3.1 AndroidManifest.xml

The AndroidManifest.xml file must be included in every application project with this exact name at the root of the project source set. The purpose of the manifest file is to describe the fundamental information about the application [54]. In addition to many other information, the manifest file must declare the following: The components of the application which include services, content providers, broadcast receivers and all activities [53]. For

better understanding of these components, see the "*Android apps*" subchapter. The permissions that an application needs to access protected parts of the system or another application. It also declares what permissions other applications must have if they want to access the content of this app. Hardware and software features the application requires, in other words, on which devices the application can be installed [54]. It should be highlighted that this is the only place where a clean or malicious application cannot "lie". Thus, if the application wants to use a certain API call that requires a particular permission that is not declared in the manifest, a Security Exception will occur. For this reason, this is where most security scanners and attacks start [54].

2.3.2 Meta-INF

The META-INF directory contains the verification information that is generated when the application is signed. It is essentially the fingerprint information for every file present in the APK. This contributes to the overall security as any modification to the APK requires re-signing otherwise the installation of this APK would be denied by the Operating System [55]. The signing process is based on the following security features. A file with the .SF suffix that contains a list of all files and their hashes [55]. The .RSA suffix file which is used to verify the integrity of the application via a public key. The MANIFEST.MF file which provides a SHA-256 hash of all files in the APK, which is used to verify the validity of every file in the zip [53].

2.3.3 Resources.arsc

The resources.arsc file provides information that directs the code (classes.dex) to the resources (res). For example, the code contains the text of the dialog while the resources contain the given text in all languages. The Android Operating System then selects the correct language based on the configuration of the device [56].

2.3.4 Classes.dex

Classes.dex is Google's proprietary format for their version of the Java virtual machine [53]. Files with the .dex suffix contain all Java/Kotlin code which is compiled into bytecode (see the "*Android Runtime*" subchapter for more details). APK files can contain more than one classes.dex due to the Android feature "Enable multidex for applications with over 64K methods" [57].

2.3.5 Res

The res folder contains resources similar to assets but with a predefined folder hierarchy that developers cannot modify. The resources are mainly the design portion of the application, such as application layout, images, strings, etc. These files are utilized to provide alternative options for varying screen orientations, Operating System versions, and multiple language support [53].

2.3.6 Assets

The asset folder contains files that are not directly used by application components. They are approached from the code via a facility known as an AssetManager. As an example, if the application displays a video or retrieves a document template, those elements will be stored in the assets folder [53].

2.3.7 Lib

The lib folder contains native libraries (machine code) [53]. Since Android is cross-platform, it contains a subfolder for each of its supported processors:

armabi - Binaries that support at least ARMv5TE, deprecated since ndkr16 (Dec. 2017) and removed in ndkr17 (Jun. 2018).

armeabi-v7a - Binaries supporting ARMv7.

arm64-v8a - Binaries supporting ARMv8.

x86 - Binaries supporting x86.

x86_64 - Binaries supporting x86-64.

mips - Binaries for MIPS, deprecated since ndkr17.

In conclusion, the Android Operating System poses several unique obstacles for the cybersecurity field as it is the most widely deployed mobile Operating System. It is due to its popularity that it has become a primary target for attackers against whom it is necessary to protect.

3 ARTIFICIAL NEURAL NETWORKS

Artificial neural networks are an increasingly favored machine learning approach imitating the learning process in biological organisms. One of the most essential components of the nervous system are neuron cells. Neurons are connected to each other by axons as well as dendrites, and the connecting regions between these moieties are called synapses [58]. Synaptic connections are capable of altering in response to an external stimulus. This change is part of a process known as "learning" and until recently was unique to living organisms. It is this biological mechanism, known as "learning", that is being mimicked in an artificial neural network containing computational units known as neurons [58]. Therefore, in this thesis, terms like neuron and neural network refers to artificial neural networks and artificial neurons not to their biological counterparts. The biological neural network and its analogue in the form of a trivial artificial neural network are depicted in Figure 6.

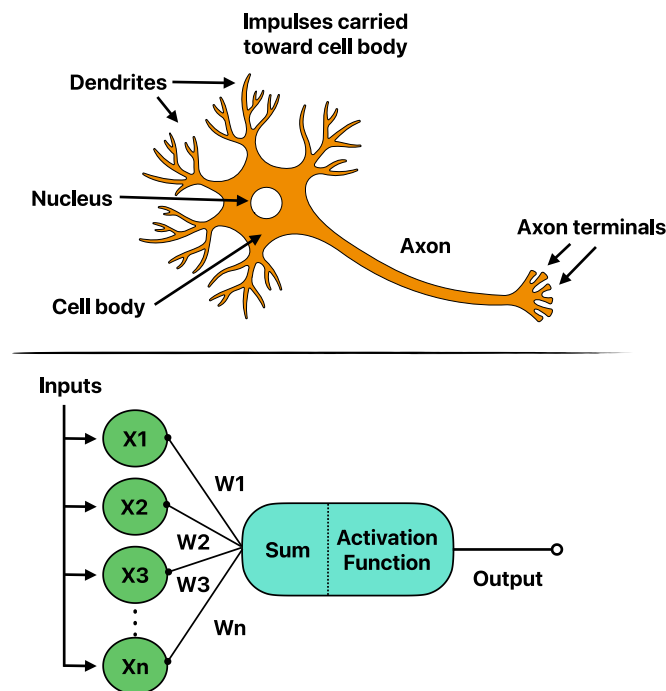


Figure 6. *Illustration of a biologic and artificial neural network.*

The connecting factor in the computational units are the weights representing the same role as the strength of the synaptic connections in the biological version [58]. The artificial neural network calculates the function of its inputs by promoting the evaluated values from the input neurons to the corresponding output neurons and by applying weights which serve as an intermediate parameter [58]. Therefore, altering the values in the weight vector can

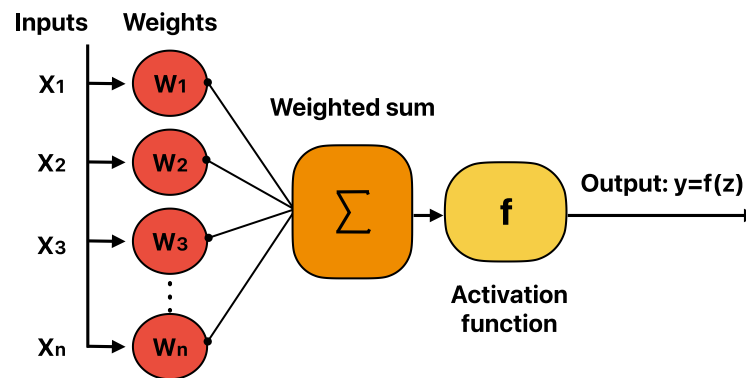
be considered "learning". Finally, just as a biological organism requires an external stimulus to learn, so do artificial neural networks. This is provided in the form of training data, containing examples of input-output parameters of the to be learned function [59]. As an example, it is possible to visualize the training data as a set of pixels representing images (greyscale or RGB depending on the complexity and type of the network) together with their labels (e.g. malware/clean application). The training data pairs are sequentially passed to the neural network using the input representation to get a prediction about the output labels [59]. Here it is necessary to mention the large variety of neural networks, which exists and where each of them is employed for a different purpose [59]. For this reason, the thesis is focused on the theory related to neural networks specialized in classification.

3.1 Architecture of neural networks

Since the early versions of neural networks, such as the perceptron, were not capable of solving complex problems, their successors address this shortcoming by applying multiple neurons arranged into layers [60]. This solution, often known as a multi-layer perceptron (MLP) or a feedforward network (as the name suggests, the individual layers feed each other in the forward direction from input to the output), allows to solve problems not only limited to linear nature. Note that a linearly separable problem can be separated using linear equation while for a nonlinear problem, the use of nonlinear equation is required [59]. In general, the architecture of these artificial neural networks (ANNs) consists of neurons, layers, weights, and an activation function [60].

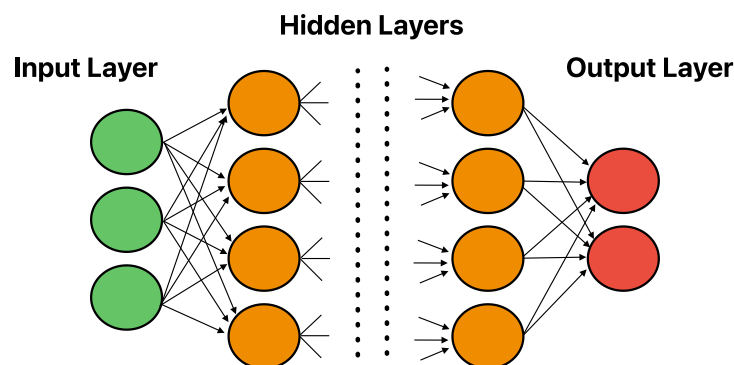
3.1.1 Neurons

The fundamental building block of every ANN is neuron. Neurons can receive one or multiple inputs depending on the input data and the number of neurons in the previous layer [60]. Another important part is the activation function within these neurons, which creates a complicated nonlinear functional mapping between input/s and output (for more details on this topic see Subsection "*Activation functions*") [60]. As can be seen in Figure 7, the neuron receives inputs $(x_1, x_2, x_3 \dots x_n)$, where the learning parameter is applied to determine a weighted sum z . The results of such an operation is then subsequently passed to the activation function f to calculate the output $f(z)$.

Figure 7. *An artificial neuron.*

3.1.2 Layer types

As stated before, the output from a single neuron $f(z)$ is insufficient to model complex tasks. Therefore, a more comprehensive structure in the form of several neurons containing layers is required [60]. A possible ANN architecture with stacked neurons into hidden layers is shown in Figure 8.

Figure 8. *Neural network architecture.*

As can be seen in Figure 8, the first section is an input layer. This entity receives data from the dataset and acts as the accessible part of the network [60]. In general, neural network diagrams assign one input value to one neuron in the input layer. The neurons situated in the input layer are used to feed the value to the upcoming layer [61]. Layers occurring after the input layer are known as hidden layers since they are not directly accessible to the input from the dataset. It is due to these structures, that ANNs are able to solve more complex problems. ANNs with more than three hidden layers are known as deep neural

networks [61]. Multiple hidden layers allow the neural network to learn data features in a so-called feature hierarchy, as initially simple features recombine from one layer to another to produce increasingly more complex features [60]. In other words, ANNs with multiple hidden layers pass input data (features) through more mathematical operations than ANNs with fewer hidden layers and thus are able to learn more complex patterns [62]. It should be noted, however, that more hidden layers (and more neurons) do not always imply a better outcome, but rather it depends on the specific issue. The last layer in the ANN is known as the output layer, and its purpose is to deliver the output value or vector of values corresponding to the format required to solve the problem (*e. g.*, whether the application is clean or malicious).

3.2 Training of neural networks

Simplistically, the training of neural networks can be summarized as the calibration of the weights of a given ANN until the desired criterion is met or the specified number of epochs is depleted [62]. The optimization itself is an iterative process (number of epochs) based on two fundamental operations: forward propagation and backpropagation, respectively. Forward propagation is a process, where input values are fed into a neural network whose output is the output value called the *predicted value*. In other words, the input values are fed into the input layer of the network (no operations are performed here, as stated in the previous subsection) which passes the values to the first hidden layer where operations such as multiplication, addition and application of the activation function are performed [58]. Subsequently, the modified values are transferred to the next hidden layer and this process is repeated until the values reach the output layer, whose result is the prediction itself. After the forward propagation is completed, the backpropagation process begins [61]. Assume that the desired output of the network is Y and the predicted value obtained using forward propagation is Y' . The difference between the predicted output and the desired output ($Y - Y'$) is transformed into the loss of the function $J(w)$ [60]. Where w represents the ANN weights. The objective of this process is to optimize the loss function (achieve the smallest possible loss) across an established training set. This optimization process utilizes a method called gradient descent. The goal of gradient descent is to find the gradient of $J(w)$ with respect to w at a given point and take a small step (learning factor) towards a negative gradient until a minimum is found [60]. A representation of the gradient descent can be seen in Figure 9.

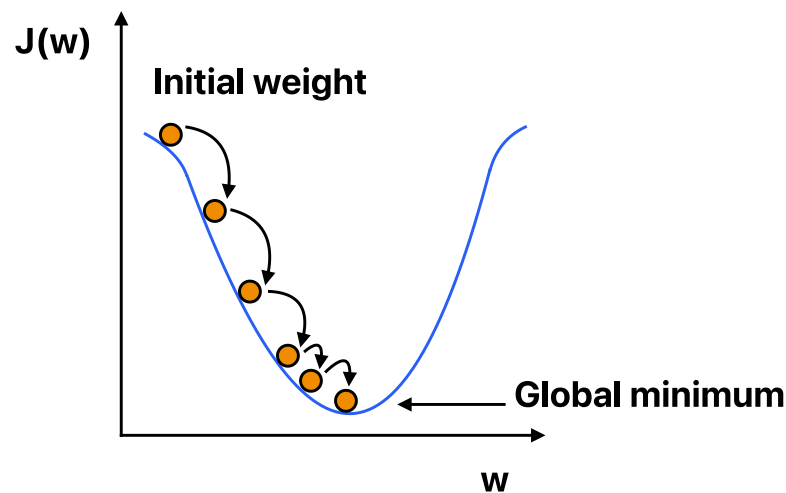


Figure 9. *Gradient descent.*

Thus, a $J(w)$ function is essentially a composition of multiple layers in a given ANN. So, if the first computational layer is represented by a function $a()$, the next layer as $b()$, and the subsequent layer as $c()$, then the objective function will assume the form: $J(w) = c(b(a()))$ where w contains the weights of all three layers [62]. The aim is to obtain the gradient of objective function $J(w)$ with respect to each weight component. In other words, the gradient of the component w in the first layer depends on the gradient of the second layer, which in turn is dependent on the gradient of the third layer [60]. Therefore, the backpropagation process, computes derivations in the reverse direction, starting from the last layer. In general, the backpropagation process is promoted from the "end" of the network layer by layer. The weights are then gradually updated based on how much they contributed to the error. It should be mentioned that almost all ANNs use this approach mainly due to its effectivity and cleanliness.

3.3 Activation functions

Another essential part of ANN is the activation function, which is applied to the weighted sum of inputs to obtain the desired output. The activation function allows the network to combine the inputs into more complex structures. This in turn allows wider capabilities in the relationships it can model and the outputs it is able to produce [58]. The activation function also determines, which of the neurons will be activated, a property that determines what information is retained for the subsequent layers. Without the availability of activation functions, ANNs would tend to lose the bulk of their representational learning abilities [59].

Overall, there is a wide range of activation functions, which are being utilized nowadays. Thus, only the most common ones are described as follows. (i) Rectified Linear Unit (ReLU) function: ReLU is represented by $f(x) = \max(x, 0)$ which ensures that if the input is positive, the output of the function is the number itself. On the contrary, if the input is negative or zero, the function returns zero. It is currently the most common activation function due to its simplicity and its potential to limit the problem of vanishing gradient [60]. (ii) Sigmoid function: The sigmoid function takes an S-shape in its graphical representation (Figure 10), is expressed using the equation $f(x) = \frac{1}{1+e^{-x}}$, and ranges from 0 to 1. This function is also known as the logistic activation function. (iii) Tanh function: The tanh activation function is in certain aspects similar to the sigmoid function. Its mathematical expression can be formulated as $f(x) = 2\text{sigmoid}(2x) - 1$, where *sigmoid* refers to the sigmoid function. The product of this activation function ranges from -1 to 1 with a uniform mass on both parts of the zero axis (Figure 10) [60]. (iv) Linear function: As the name of the activation function suggests, its representation is given by a straight-line equation $f(x) = mx + c$, where the activation is proportional to the output [60]. It should be mentioned that in a case the network contains multiple layers and all layers are linear in nature, the final activation function will take the same format as the first one. The output of this activation function ranges from $-\infty$ to ∞ .

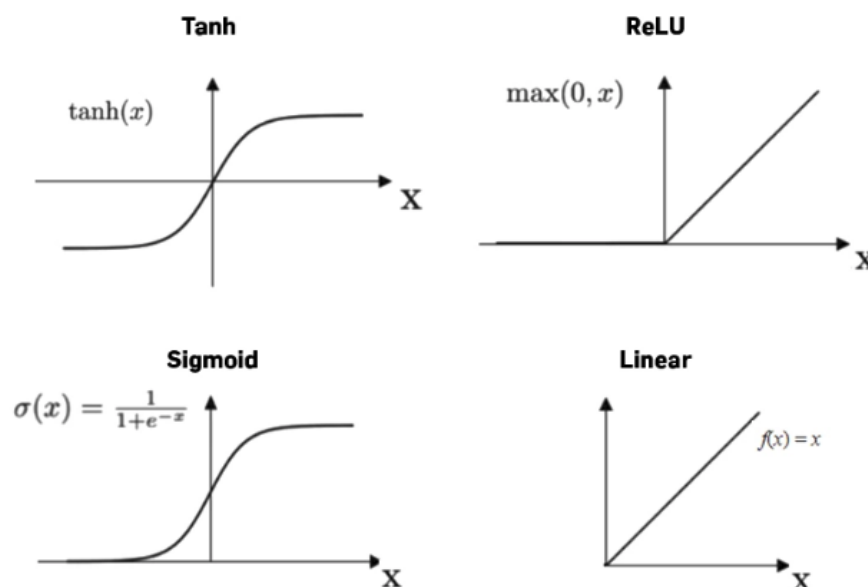


Figure 10. Activation functions [50].

3.4 Convolution neural networks

Convolutional neural networks, similarly, to perceptron or feedforward networks, are also biologically inspired. In addition, they are currently utilized in substantial applications across a variety of fields, namely computer vision or image classification. The architecture of convolutional neural networks (CNN) is unique in the sense that each layer is three-dimensional, whose spatial extension and depth correspond to the number of features [59]. However, it is essential to note that the depth of a specific layer in a CNN is different from the notion of depth related to the number of layers. For instance, the features occurring in the input layer match the color channels, most commonly RGB, and in the hidden channels these features represent hidden feature maps encoding the different types of shapes presented in the image. A useful illustration provides the input to a convolutional neural network in a form of grayscale image. In this situation, the depth of the input will be equal to one, but subsequent layers will be three-dimensional [59]. The convolution operation, using filters to map activation from one layer to another, represents one of the key techniques underlying convolutional neural layers. The convolution operation utilizes three-dimensional filter of weights that have the same dimension of the active layer, but with a reduced spatial extent [62]. The value of the hidden state in the upcoming layer is calculated as the dot product of all weights in the filter and any choice of spatial region (must be the same size as the filter) in the layer (after applying the activation function). This operation is performed among the filter and the spatial regions in a given layer for all possible locations to determine the next layer [62]. Representation of the convolution operation is presented in Figure 11.

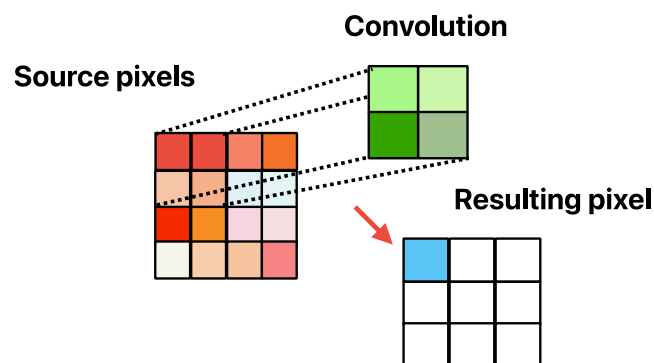


Figure 11. *Convolution operation.*

Pooling is another essential operation frequently utilized in CNN. The general concept behind the pooling layer is to accumulate features from maps produced by convolutional filters over the given image [60]. The purpose of the pooling layer is to gradually reduce the spatial size of the representation to decrease the number of parameters and the computation power needed by a given CNN [60]. The most common variant of pooling operation is max pooling, which helps to reduce over-fitting. The max pooling operation is based on applying a max filter to non-overlapping subregions of the initial representation. Naturally, there are other variations of pooling, such as general or average pooling. The scheme illustrating the process of max pooling operation is depicted in Figure 12.

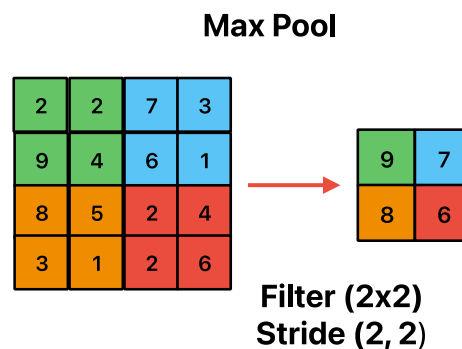


Figure 12. *Max pooling operation.*

It is important to note that the connections in a convolutional neural network are fairly sparse. This is due to the fact that the activation in a given layer is a function of the tiny spatial region of the preceding layers [59]. Thus, with the exception of the last two or three layers, all layers retain their spatial structure. Such a feature allows to spatially display which parts of the image influence specific activation positions of a given layer. Generally speaking, the initial layers can only process trivial shapes such as lines. The "deeper" convolutional neural network, however, can tackle more complex shapes [62]. After applying convolutional and pooling layers in a given CNN (additional layers such as batch normalization should be utilized), the transformed features are fed to the fully connected layer. A fully connected layer takes features from previous layers, now flattened (one dimensional), and then processes them as normal ANN. It is therefore possible to specify already known hyperparameters, such as the number of hidden layers or the number of neurons in these layers [62]. Finally, the CNN contains an output layer, which results in the classification itself (clean/malicious application). The layers

and methodologies discussed above form the building blocks for the CNN architecture as such (Figure 13).

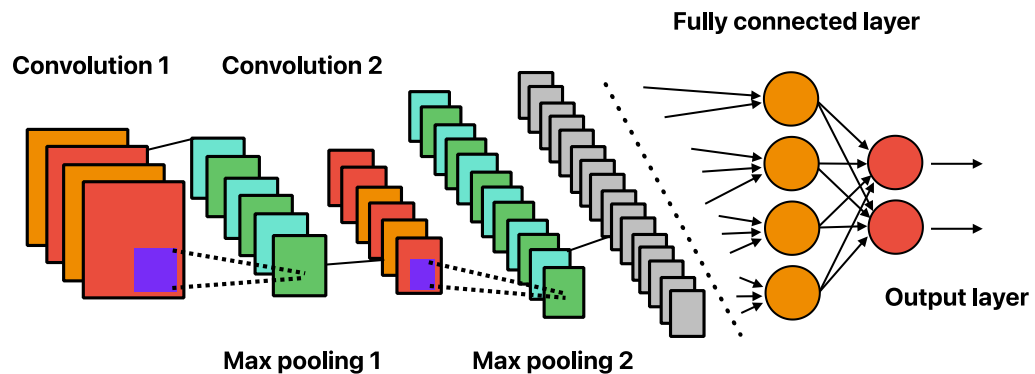


Figure 13. *Architecture of CNN.*

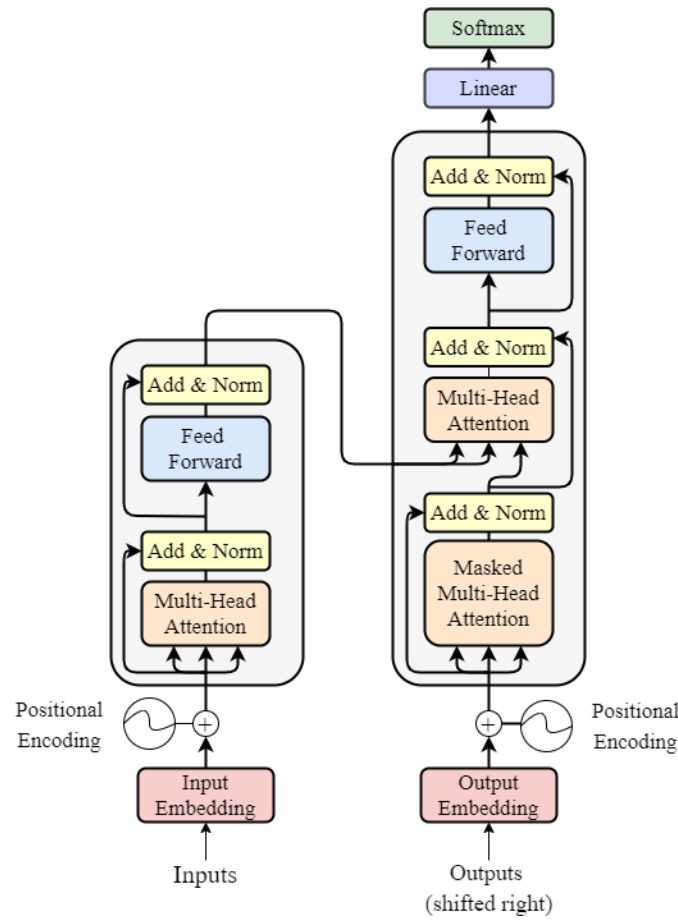
Please note that this subsection describes only the very basics of CNN and therefore it is recommended to study operations such as batch normalization, dropout etc. before going through the practical section. That said, CNNs have revolutionized the field of computer vision by enhancing image recognition, object detection, and image segmentation tasks, respectively. The achievements of CNNs can be contributed mainly to their ability to automatically learn the features present in an image using fundamental operations such as convolution or max pooling [62]. Hence, this thesis will also employ this approach to the problem of Android malware detection for image-shaped features. However, due to the complex nature of the Android malware detection problem, it is likely that the features extracted from the APK will not only be in the form of an image, but also include ancillary features such as a binary vector. For this reason, it is recommended to use a combination of machine learning methods (as mentioned in the chapter "*RESEARCH INTO THE PROBLEM OF DETECTING MALICIOUS APPLICATIONS ON THE ANDROID PLATFORM USING MACHINE LEARNING METHODS*") and thus, from a neural network perspective, it is possible to combine *e.g.* CNN and MLP for features of different forms.

4 TRANSFORMERS

Transformers were firstly introduced as a part of the machine learning subfield called as natural language processing (NLP) in 2017 in the scientific paper "*Attention is all you need*". Since then, they have started to prevail over conventional neural networks like Long Short-Term memory (LSTM) or Gated Recurrent Unit (GRU) [26]. The employment of transformers in the field of computer vision, on the other hand, is still hindered due to the widespread use of convolutional neural networks. One of the ground-breaking attempts to apply the transformer architecture to the problem of image recognition was presented in the research article "*An Image is Worth 16 x 16 Words: Transformers for Image Recognition at Scale*", exploiting a Vision Transformer (ViT) model [27]. Transformers are networks functioning on a sequence of data, *e.g.* instance a set of words. These sequences are initially tokenized and then used as input to the transformer [26]. The transformers utilize a unique operation called attention, which computes the inner pairwise product among each pair of tokenized words. To understand the special case of a transformer, called ViT, it is first necessary to understand the traditional architecture dealing with the NLP domain.

4.1 Model Architecture

In general, the majority of credible models for NLP contain two basic elements: an encoder and a decoder [26]. The encoder maps an input sequence represented as $(x_1, x_2, x_3 \dots x_n)$ into a series of continuous representation $z = (z_1, z_2, z_3 \dots z_n)$. The decoder then generates the output sequence $(y_1, y_2, y_3 \dots y_n)$ of symbols from the given z in a sequential step (one element after another) [26]. Here it is necessary to note, that the discussed model is auto-regressive, implying it utilizes the previously generated symbols as an additional input to generate the subsequent symbol. As mentioned before, the transformer architecture uses stacked attention and pointwise fully connected layers for both encoder and decoder [26]. The overall architecture of the baseline version of the transformer is shown in Figure 14. The most relevant components and operations such as the previously mentioned encoder, decoder, and attention will be sequentially described and explained in following chapters. This will form the basis for the theory of the modified version of the transformer (ViT) [27].

Figure 14. *Transformer architecture* [23].

4.1.1 Encoder and Decoder

Encoder: The encoder in this example is composed of a stack of $N = 6$ identical layers. Where each layer contains two sublayers, *i.e.* the first consisting of the self-attention mechanism and the second possessing the position-wise fully connected feed-forward network [26]. Next, the authors of the papers employ residual connections around each of the two sub-layers, followed by a normalization layer. This implies, that the output of each sub-layer is $\text{LayerNorm}(x + \text{Sublayer}(x))$, where $\text{Sublayer}(x)$ is a function implemented by the sub-layer itself [26].

Decoder: The decoder also consists of a stack of $N = 6$ identical layers [26]. Contrary to the encoder, containing only two sub-layers for each layer, the decoder includes a third layer, which performs a multi-head attention *via* the output of the encoder stack. Subsequently, residual connections are applied around each sublayer followed by a normalization layer [26]. Lastly, the authors of the paper modified the self-attention sub-layer in the decoder stack to avoid positions from attending to the subsequent positions [26].

4.1.2 Attention

An attention function can be characterized as mapping of a query and a set of key-value pairs to an output, where the query, keys, values, and output are vectors [26]. The output is therefore obtained as a weighted sum of values, with the weight of each value computed using the query compatibility function with the corresponding key [26]. A representation of multi-head attention and scaled dot-product attention is shown in Figure 15.

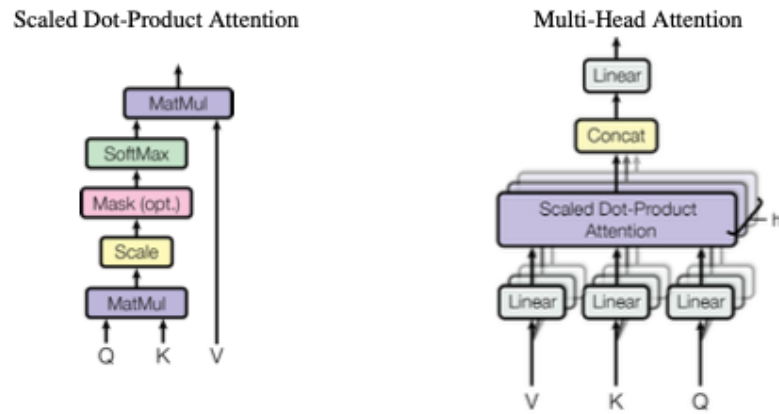


Figure 15. *Scaled Dot-Product Attention and Multi-Head Attention* [26].

(i) Scaled Dot-Product Attention: As discussed above, the input is made up of queries and keys of dimensions d_k and values of dimensions d_v . Here, the dot product is calculated using the query with all keys and divided by each $\sqrt{d_k}$, followed by applying the SoftMax function to obtain the weights for the respective values [26]. For practical reasons, the attention function is calculated on a set of queries at the same time, wrapped together in a matrix Q . Like the queries, the keys and values are also in the form of matrices K and V [26]. The output matrix can therefore be computed using equation (1).

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad (1)$$

(ii) Multi-Head Attention: Although attention itself can be considered as a high-potential operation, the authors of this paper observed that instead of a single attention function with d_{model} - dimensional keys, values, and queries, it is more efficient to linearly project queries, keys, and values h times with various learned linear projections to d_k , d_k and d_v dimensions [26]. Subsequently, for each of these projected versions of queries, keys, and values, the attention function is applied in parallel resulting in a d_v -dimensional output of values [26]. In general, multiple-headed attention permits the model to jointly attend to information

from various representation spaces at distinct positions [26]. Please note that this subsection does not explain the entirety of the baseline version of the transformer, but rather serves as a preliminary preparation for ViT.

4.2 Vision Transformer (ViT)

Since the baseline version of the transformer is designed for NLP or similar problems, it is not suitable for our classification of a malicious Android app (as further discussed in the chapter *Research into the problem of detecting malicious applications on the Android platform using machine learning methods*). For this reason, the thesis is focused on a special case of transformer (ViT), which primarily deals with image processing. The problem arises and especially in the issue of Android application malware detection, that the image composed of features extracted from APKs reaches considerable dimensions [27]. To illustrate this issue, consider an image with size of $250 * 250$ pixels (250^2), the attention mechanism would then cost $(250^2)^2$ operations (due to its quadratic nature). Such a complex and expensive task poses a challenge even for high-performance GPUs. Therefore, for high-resolution images, a special kind of attention, called local attention (pixel clustering), is often applied to reduce its computational complexity [27]. The ViT architecture is subsequently described in Figure 16.

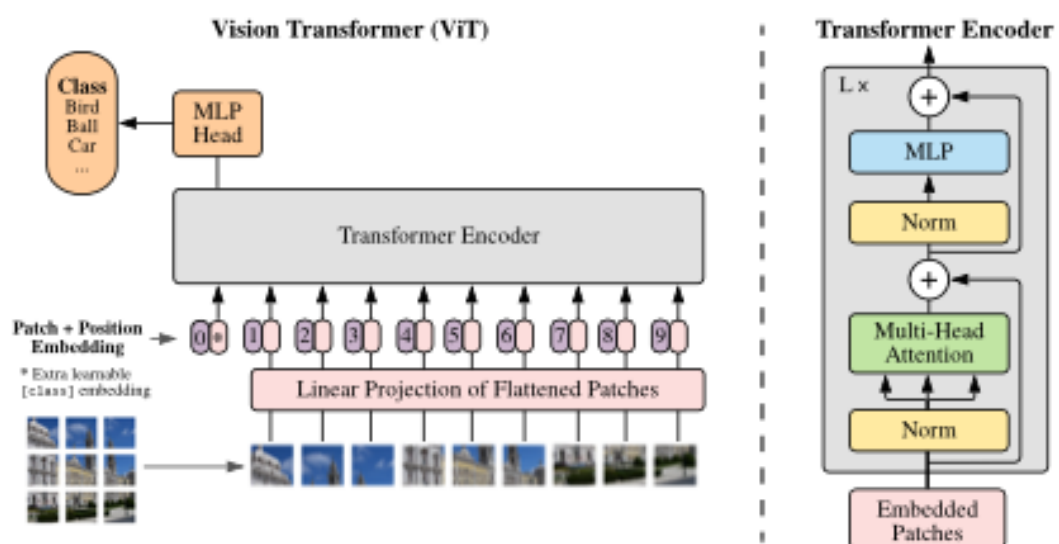


Figure 16. *Vision Transformer (ViT)* [27].

In the following paragraph the overall function of ViT will be dissected into individual sub-operations, which will be elaborated in detail. The first step is to split the image into fixed-size patches and embed each of them linearly. Next, it is necessary to add positional

embedding and pass the resulting sequence of vectors to a standard transformer [27]. The transformer then receives the passed input in the form of a one-dimensional sequence of token embeddings. To be able to process two-dimensional images, it is necessary to reshape the image $x \in \mathbb{R}^{H \times W \times C}$ into a sequence of flattened 2D patches $x_p \in \mathbb{R}^{N \times (P^2 C)}$, where (H, W) is the resolution of the original image [27]. Furthermore, C denotes the number of channels, (P, P) represents the resolution of each image in the patch, and $N = HW/P^2$ is the resulting number of patches, which also acts as the length of the input sequence for the transformer [27]. Here it should be mentioned that the transformer uses the constant latent vector size D through all its layers. Thus, it is necessary to flatten the patches and map to D dimension with a trainable linear projection (equation 2) [27].

$$z_0 = [x_{class}; x_p^1 E; x_p^2 E, \dots x_p^N E] + E_{pos}, \quad E \in \mathbb{R}^{(P^2 C) \times D}, E_{pos} \in \mathbb{R}^{(N+1) \times D} \quad (2)$$

Furthermore, it is necessary to transfer the learnable embedding to the sequence of embedded patches $z_0^0 = x_{class}$, whose state at the output of the encoder z_L^0 acts as a representation of the image y [27]. For both the pre-training and fine-tuning phase, a classification head is linked to the encoder. The classification head is a basic MLP with one hidden layer for the pre-training phase and one linear layer for the fine-tuning phase (for more information on MLP, see the chapter “*ARTIFICIAL NEURAL NETWORKS*”). Moreover, position embeddings are included to the patch embeddings to preserve positional information. According to the authors, there is no need to use more complex 2D-aware positional embeddings as they achieve similar results to standard 1D positional embeddings [27]. The encoder architecture itself is the same as in the subsection “*model architecture*”, where the only additional information is the fact that MLP uses the GELU activation function [27].

In conclusion, although this new method is in its early development stage in analyzing large amounts of data, it often provides the same results as the best CNN [27]. For this reason, this methodology will also be experimentally employed for the Android malware detection problem.

5 HYPER-PARAMETER OPTIMIZATION

With the Android OS architecture explained and machine learning methods discussed, it is appropriate to devote the last chapter of the theoretical section to the optimization of hyper-parameters. The objective of the hyper-parameter optimization for a given machine learning algorithm is to determine the "optimal" parameters for a given architecture to achieve the best performance on the test set [63]. Here it should be specified, that there are two main types of parameters: (i) model parameters, *e.g.* weights in neural networks, optimized by the model during the learning process; (ii) hyper-parameters, such as the number of hidden layers, which are determined by the user [64]. In general, hyper-parameters can be envisioned as model settings that need to be fine-tuned to ensure an optimal solution to the problem. The possible hyper-parameters of the described machine learning methods are listed in Table 5.

Table 5. *Illustration of possible hyper-parameters.*

ML method	Hyper-parameters
MLP	Number of hidden layers
	Number of neurons in hidden layer
	Activation function
CNN	Kernel size
	Dropout coefficient
	Stride and padding
ViT	Number of heads
	Patch size
	Learning rate

5.1 Methods of hyper-optimization

It is hardly surprising that the choice of hyper-parameters has a significant impact on the learning of ML algorithms. Therefore, there is a wide range of methodologies addressing this issue, which are also commonly employed in practice. (i) Manual Hyperparameter Tuning: The most trivial way to optimize hyper-parameters is manually,

in other words, by trial-and-error. This methodology is not based on any algorithm. It rather relies on the expertise of a given specialist, who is able to "guess" the right parameters to achieve high accuracy [65]. Even though the experience in this field is essential, this methodology is recommended only for simple issues and not for complex ones (*e. g.* detection of Android malware). (ii) Grid Search: Grid search is another elementary technique for identifying the optimal setup. Similar, to the previous manual tuning, it is based on imposing constraints on each of the hyperparameters, rather than any complex algorithm. For example, it is possible to specify that a hidden layer H_l can contain $N_{neurons} = \{5, 10, 15\}$ neurons where for each variant within a given grid, the model is tested. Even here it is possible to notice the shortcomings of the given methodology at first glance. Namely, the optimal solution can be situated outside the specified boundaries and the actual testing for more complex algorithms can be excessively long [66]. (iii) Random Search: Often certain hyper-parameters have a greater impact on the outcome than others. The use of random search compared to grid search allows the algorithm more flexibility in terms of the searched space and therefore a better chance of finding a better result. Random search, as the name suggests, randomly selects a combination of chosen hyperparameters for a given number of optimization cycles. In theory, with an infinite amount of time, a random search will always find the optimal solution. However, just as previous method, random search approach is also hindered by the long learning time required for complex models. Thus, this method is still considered as inefficient [65]. Naturally, there are numerous other methods such as the hill-climber algorithm, simulated annealing, genetic algorithms, particle swarm optimization and others. However, due to the specific focus of this thesis, these algorithms will not be discussed.

5.2 Bayesian Optimization

Bayesian optimization, unlike the approaches discussed, takes into consideration the previously evaluated solutions, and uses them to construct a probabilistic model. Such a model maps the given hyperparameters to a score probability on the objective function: $P(score | hyperparameters)$ [63]. In general, this type of model is known as a surrogate for the objective function and is represented as $p(y | x)$. A significant advantage is that the surrogate is usually much easier to optimize compared to the objective function (especially for complex models) [62]. Bayesian methods therefore work by finding another set of hyperparameters to evaluate the objective function by selecting the hyperparameters best suited to the surrogate function. In other words, the first step is to construct a surrogate

probabilistic model of the objective function [65]. Next, it is necessary to find a set of hyperparameters leading to the best performance on the surrogate function. Subsequently, these hyperparameters are applied to the objective function, which allows to update the surrogate model by incorporating new results. This process is repeated until a given criterion is satisfied or the number of optimization cycles is reached. Due to this procedure, Bayesian models can be regarded as efficient especially for complex models, because they determine the posterior hyperparameters in an informed manner. The general idea behind this process is to spend more time on the selection of hyperparameters in order to make less evaluation of the objective function. Such an approach can save a significant amount of time [64].

5.3 Tree-Structured Parzen Estimator

A tree-structured Parzen Estimator algorithm is developed to optimize the quantized hyperparameters to obtain the configuration yielding the expected outcome and providing the highest possible latency improvement [62]. Tree-structured Parzen Estimator is an iterative process that uses the previously estimated hyperparameters to form a probabilistic model, that is utilized to obtain the subsequent set of hyperparameters [62]. Simplistically, this algorithm can be summarized in the following steps: (i) Definition of the domain for the hyperparameter search space. (ii) Creating an objective function that takes hyper-parameters as inputs and produces a score (*e.g.*, mean square error) that needs to be minimized [65]. (iii) Initialize a few randomly selected sets of hyperparameters to obtain an initial observation (scores). (iv) Sort the collected observations by their scores and then divide them into two groups based on the quantile (the first group contains the best results and the second contains the others). (v) Construct two densities, known as kernel density estimators, which are a simple average of kernels centered on existing data points [64]. (vi) Sample hyperparameters from the "successful" density to rank them as $D(x_s) / D(x_o)$ and return the set that shows the minimum value according to the largest possible improvement. This set of hyperparameters is then tested on the objective function. (vii) Update the observation list in step (iii) and repeat steps (iv)-(vi) until the criterion is met or the number of optimization cycles is exhausted [64].

In conclusion, hyperparameter optimization is an essential step in the process of testing a machine learning model. That is, by systematically exploring the hyperparameter space allowing to find the optimal configuration for a given task [63].

II. ANALYSIS

6 DATA GATHERING

Chapter “RESEARCH INTO THE PROBLEM OF DETECTING MALICIOUS APPLICATIONS ON THE ANDROID PLATFORM USING MACHINE LEARNING METHODS” together with the rest of theoretical part of the thesis provides an introduction to the fundamentals of detecting malware running under the Android operating system. The next step towards developing a case study using machine learning methods is to gather data from which a comprehensive dataset can be formed. In general, the dataset should contain a sufficient amount of data, i.e. clean and malicious applications, so that the applied machine learning methods are capable of their efficient classification. The first source of Android applications in this thesis represents the academic dataset CICMalDroid 2020 [67] which contains several groups of malicious applications such as banking malware, SMS malware and Adware. Another source of data for the establishment of the dataset is the Penetration Testing Laboratory (PT LAB) [68] located at the University of Tomas Bata in Zlín. It should be noted that throughout the thesis, the applications are referred to as clean or malicious. Table 6 shows an overview of the dataset produced including the sources from which the applications were collected.

Table 6. Dataset overview

Dataset		
Source	Application type	n. application
CICMalDroid 2020	Adware	424
	Banking malware	424
	SMS malware	423
	Riskware	424
PT LAB	Clean	1696
		$\Sigma = 3391$

The total number of Android applications, with the same proportion of both groups, was set to 3391 (Table 6). The chosen number of applications represents a reasonable balance between a sufficient number of samples for accurate classification and the technical

limitations inherent in the complexity of machine learning models. To obtain a basic insight into the collected data, a fundamental analysis of the clean and malicious applications size was performed (Figure 17.).

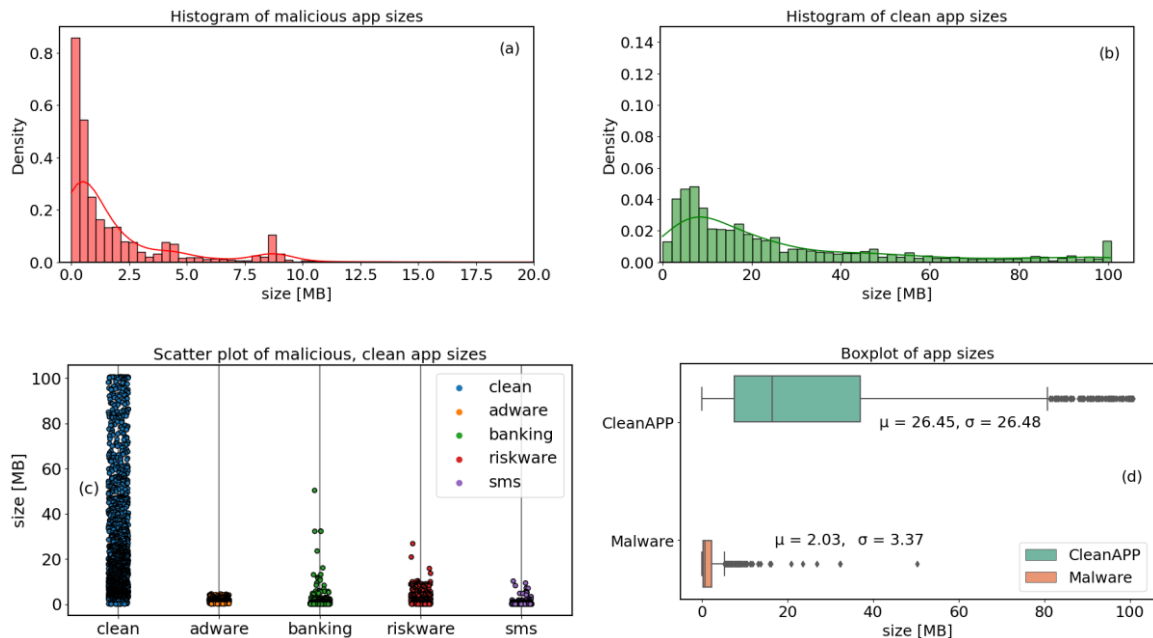


Figure 17. Size analysis of the clean and malicious applications.

The size of Android clean and malicious applications was dissected in four different points of view, sequentially described below. Figure 17a shows a size distribution histogram of malicious applications size in megabytes [MB], where it is possible to observe the distribution of values into bins. Here, it is important to note that the set of malicious applications exhibited a small number of individuals with size outside the selected interval for this histogram, but for readability reasons they were not included in Figure 17a. Furthermore, it can be seen that the size of most malicious applications ranged from 1 to 5 MB. However, here it is important to state that this observation standalone just provides an information about the presented data set behavior and does not imply that the malicious applications are generally smaller in size compared to the clean ones. Figure 17b shows a histogram of clean application sizes where the distribution of values can be considered more "uniform" although application sizes ranging from 1 to 20 MB are still a dominant feature.

Figure 17c is a scatter plot where applications are divided into four subgroups namely clean, adware, banking malware, riskware and SMS malware applications and are represented

by a point which shows the size of the application of that subgroup. A jitter function is implemented in the graph, which partially prevents overlapping to simplify readability. Also here, a higher degree of uniformity of the distribution of clean applications can be observed. Finally, Figure 17d represents the size of the clean and malicious applications using a boxplot, which shows the basic statistical characteristics such as minimum value, maximum value, median, mean, 25th percentile, 75th percentile, and outliers. Below each boxplot, the mean and standard deviation values are highlighted to get a general idea of value range.

Splitting of the dataset into subsets

To achieve robust results, a proper partitioning of the created dataset is required. At this point in time, the dataset is made up of three 3391 applications in almost equal parts of each of the two groups. The goal of this subsection is to create subsets of the dataset that allow for superior learning of the machine learning models. In general, the use of training, testing and validation sets is recommended in the process of "teaching" machine learning models. Figure 18. depicts the best practices for dataset partitioning.

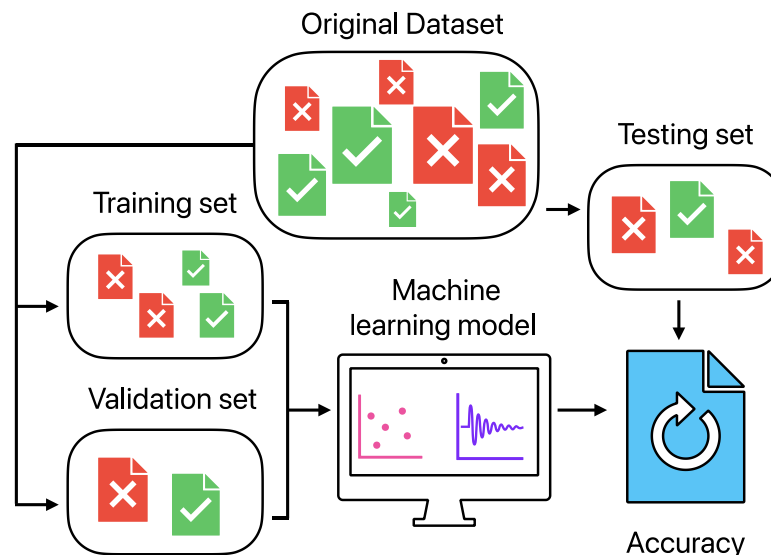


Figure 18. Splitting of the dataset.

Training set: The training set is used as the main source of learning, (fitting) the model. The dataset in this case the collection of clean and malicious applications should form proportionally the largest part of all auxiliary sets. An unwritten rule of thumb in machine learning circles, based on experience, states the size of the training set to be between 60-80% of the original dataset and this study follows this guideline as well.

Testing set: The test set serves as a reflection of the quality of the model that was fit on the training set. This step is necessary to get "true" accuracy on data that the created model did not have access to during training. The main reason to perform this step is that a significant portion of machine learning methods suffer from overfitting. The size of this set for this study is determined to be 20% of the original dataset.

Validation set: The validation set, which can be considered as a special case of the test set, is formed from a separated portion of the training set (in this case study consider 25%, which the model cannot access during learning and serves to prevent overfitting during the training phase of the cycle.

To specify the properties of the described subsets, a summary of their sizes is provided in Table 7.

Table 7. Subset summary.

Dataset type	n. application
Original dataset	3391
Train set	2170
Test set	679
Validation set	542

7 FEATURE EXTRACTION FROM AN ANDROID APPLICATION WITH DETECTION POTENTIAL

With the proper partitioning of the dataset described in the previous chapter, it is necessary to proceed with the preprocessing of the Android applications. The aim of this chapter is to describe the extracted features with the detection potential of the Android malware. Table 3. in the *RESEARCH INTO THE PROBLEM OF DETECTING MALICIOUS APPLICATIONS ON THE ANDROID PLATFORM USING MACHINE LEARNING METHODS* " chapter provides an overview of the scientific papers that inspired the feature extraction process itself. It should be noted that the following subchapters contain pseudocodes, that do not represent any programming language, nor can they be considered functional. Thus, the purpose of these pseudocodes is purely informative. An outline of the essential technologies used for extraction of the features with detection potential is shown in Table 8.

Table 8. A summary of feature extraction technologies for Android applications.

Technology	Description
python	programming language
androguard	python tool for working with Android files
dexparser	powerful DEX file format parser
pandas	library for data handling
numpy	scientific computing with python

Each methodology of feature extraction with detection potential will be introduced and described. Moreover, it will be followed by a detailed breakdown of the procedure that was implemented. At the end the outputs that can be obtained by the methodology will be evaluated and presented.

7.1 APK to greyscale image approach

The first approach is inspired by the paper "*Efficient Malware Classification by Binary Sequences with One-Dimensional Convolutional Neural Networks [20]*" where the Android application is viewed as a sequence of bytes that can be represented as a grayscale image.

To get a better understanding of such an approach, Figure 19 provides a simplified view of the main steps.

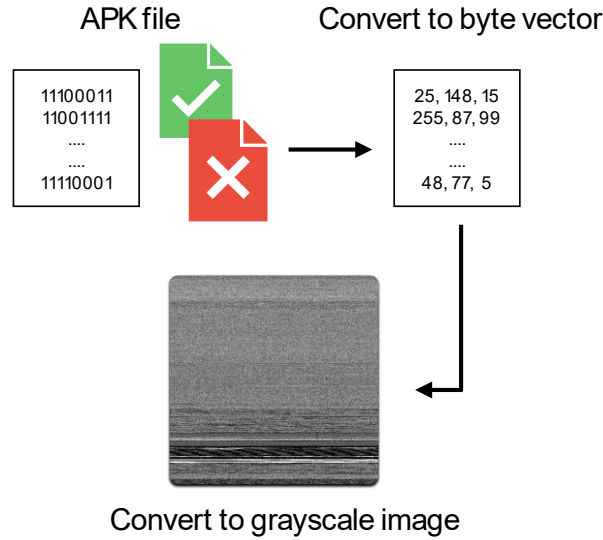


Figure 19. Simplified methodology of APK to greyscale image.

Figure 19 merely illustrates the main points of the methodology in generic terms, and therefore a detailed description of the approach follows. Table 9 serves as an auxiliary table which represents the pseudo-code for the process of converting an Android application to a greyscale image.

Table 9. Pseudo code that represents the conversion of an Android application to a greyscale image.

Algorithm 1

For *Apk* in *ApkDataset*:

ByteVector = *ReadBytes*(*Apk*)

Width, Height = *CalculateImageShape*(*ByteVector*)

ByteVector = *AddZeroPadding*(*ByteVector*)

ByteVector = *ReshapeVector*(*ByteVector*, *Width*, *Height*)

SaveAsGreyscaleImage(*PathToSaveImage*, *ByteVector*)

As shown in Table 9, the first step is to load all bytes of the iterated application. In this step, one-dimensional array V of size N (depending on the size of the application) is produced.

In order to create a grayscale image, it is necessary to change the shape of the V array from one-dimensional to two-dimensional. For this purpose, the subsequent step is performed, where the height H and width W of the image are calculated from the initial size N of the array V using formula $H = W = \lceil \sqrt{N} \rceil$. Furthermore, it is necessary to add an intermediate step in the form of zero padding, which ensures that the shape of the resulting image has a balanced characteristics of height H and width W . Next, the array V itself is reshaped to the dimensions of the calculated height H and width W . Finally, the array V is saved as a grayscale image in the selected directory. These steps are repeated for each application in the created dataset.

The result of such pre-processing of Android applications is a set of grayscale images both for clean and malicious applications that can serve as input to machine learning models. Figure 20 is used as an illustration of randomly selected applications from both groups that were preprocessed using the "APK to grayscale image" methodology.

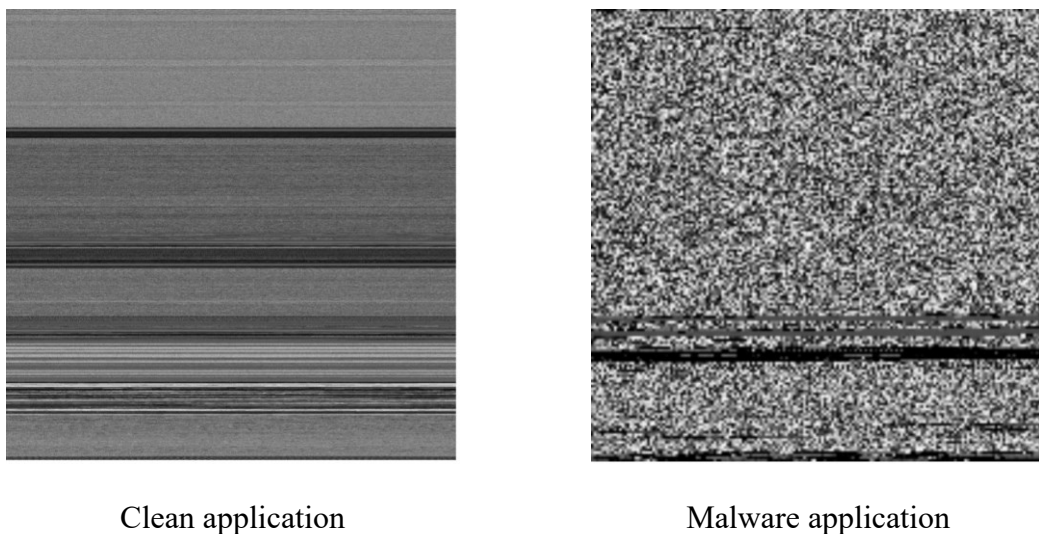


Figure 20. Samples of grayscale images for "APK to grayscale image" approach.

7.2 DEX to greyscale image approach

Another variant of looking at an Android application as an image is influenced by the scientific paper "Android Malware Detection using Convolutional Neural Networks and Data Section Images" [22]. The methodology proposed by the authors involves loading the APK and then extracting only the classes.dex and not the application as a whole, see the previous approach. Subsequently, only the data portion of classes.dex is extracted as a sequence of bytes, which is then transformed into a greyscale image. The overall extraction process is shown in Figure 21.

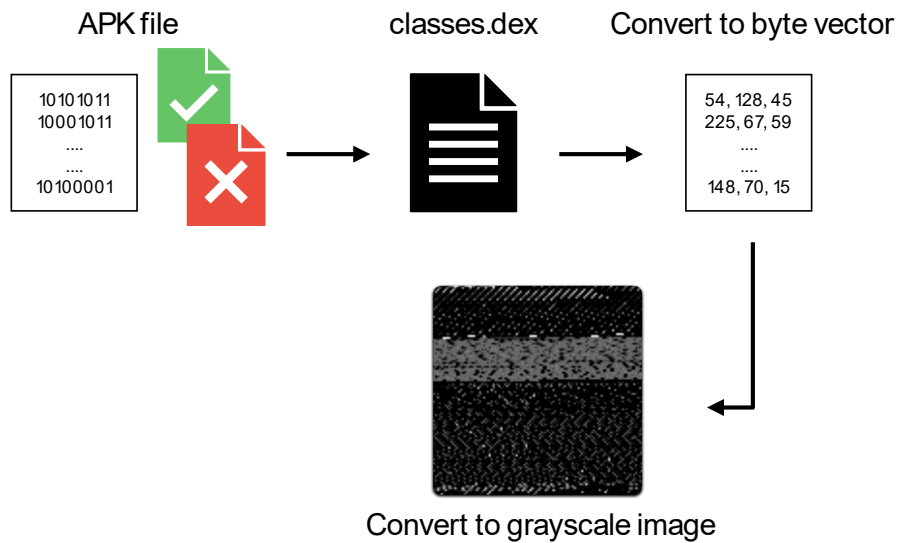


Figure 21. simplified methodology of DEX to greyscale image.

In this approach, it is considered that Android applications contain only one classes.dex which is rather exceptional nowadays and therefore in this research all dex files were extracted. Further, the authors of the paper extract only the data portion of classes.dex whereas in the case of this thesis all sections of classes.dex were used due to the relevant features found there. The modified procedure is shown in Table 10 in the form of pseudocode.

Table 10. Pseudo code that represents the conversion of DEX to a grayscale image.

Algorithm 2

ByteVector = *Array()*

For *Apk* *in* *ApkDataset*:

For *Dexes* *in* *Apk*:

SectionVector = *ReadBytesOfAllSections(Dexes)*

ByteVector = *ConcatenateSections(ByteVector, SectionVector)*

Width, Height = *CalculateImageShape(ByteVector)*

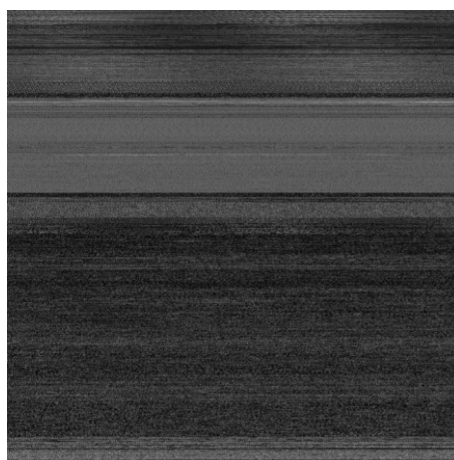
ByteVector = *AddZeroPadding(ByteVector)*

ByteVector = *ReshapeVector(ByteVector, Width, Height)*

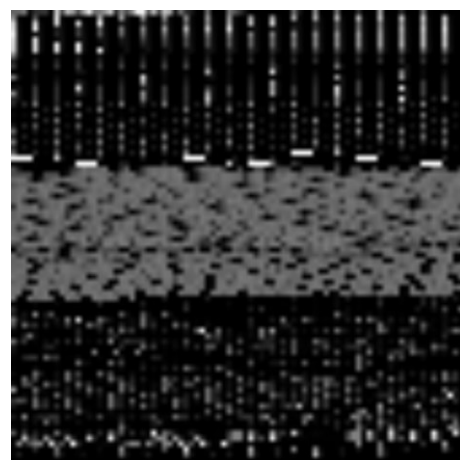
SaveAsGreyScaleImage(PathToSaveImage, ByteVector)

The first step after loading the iterated application is to assign all the section bytes from selected `classes.dex` to the array S . Furthermore, matching sections from different `classes.dex` are concatenated into the array V which contains all section bytes of all `classes.dex` in the application. Subsequently, as in the previous method, the height H and width W are calculated using the size N to determine the image dimension. Also, the zero-padding function is implemented to ensure the balanced characteristics of the shape of the grayscale image. Next, the array V is reshaped according to the calculated height H and width W into a two-dimensional array. The last step is to save the array V as a grayscale image to the desired directory. This procedure is repeated for all Android applications in the dataset.

The chosen methodology leads to a set of grayscale images that are created from all `classes.dex` that the Android applications contained. In contrast to the previous approach, the image sizes can be regarded as smaller because the bytes of the entire application are not loaded, but only of all `classes.dex`. Figure 22 shows two randomly selected applications from the created dataset that were preprocessed using this method.



Clean application



Malware application

Figure 22. Samples of grayscale images for "DEX to grayscale image" approach.

7.3 DEX to RGB image approach

The last approach of converting an Android application to an image which is addressed in presented master thesis is based on the research paper "*Android Malware Familial Classification Based on DEX File Section Features*" [23]. As in the previous method, the authors of the paper first extract the `classes.dex` files from the Android application, but as in the modified pseudocode (Table 10), they take into account all sections of `classes.dex`.

After reading the classes.dex files as a sequence of bytes, in addition to each section, the entropy (S) is calculated using equation (3) and the proportion of the section using equation (4).

$$Entropy = \sum_{i=0}^{255} p(c_i) \log_2 p(c_i) \quad (3)$$

Where c is the number of occurrences of byte i and $p(c_i)$ represents the probability of occurrence of the byte. The results of such an equation are entropy values ranging in the interval $\langle 0,8 \rangle$ since the output of this methodology is an image the range of values is then transformed to the interval $\langle 0,255 \rangle$ to match the pixel values.

$$Proportion = \frac{SectionSize}{FileSize} \quad (4)$$

Similar to equation (3), the proportion (4) requires converting the range of values to the interval $\langle 0,255 \rangle$. After supplementing the entropy and proportion values to match the dimensions of the sequence of bytes, it is possible to create an RGB (Red, Green, Blue) image where each component of the image corresponds to one dimension of the created matrix. A simplified representation of the process is shown in Figure 23.

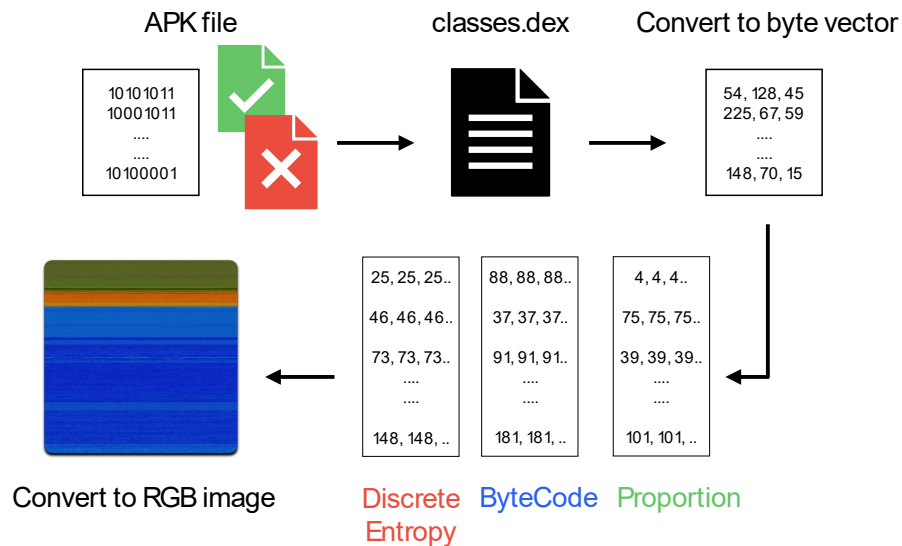


Figure 23. Simplified methodology of DEX to RGB image.

Now with the fundamentals of the selected approach, it is possible to proceed to its exact description, which is shown in Table 11 as pseudocode.

Table 11. Pseudo code that represents the conversion of DEX to a RGB image.

Algorithm 3

```
ByteVector = Array()
```

```
For Apk in ApkDataset:
```

```
    For Dexes in Apk:
```

```
        SectionBytes = ReadBytesOfAllSections(Dexes)
```

```
        SectionEntropy = CalculateEntropyOfSections(SectionBytes, eq(1))
```

```
        SectionProportion = CalculateProportionOfSections(SectionBytes, eq(2))
```

```
        AddPadding(len(SectionBytes), SectionEntropy, SectionProportion)
```

```
        ByteVector = ConcatenateSections(
```

```
            ByteVector, SectionBytes, SectionEntropy, SectionProportion)
```

```
        Width, Height = CalculateImageShape(ByteVector)
```

```
        ByteVector = AddZeroPadding(ByteVector)
```

```
        ByteVector = ReshapeVector(ByteVector, Width, Height)
```

```
        SaveAsRGBImage(PathToSaveImage, ByteVector)
```

The first step after loading the iterated application is to read all the bytes of every classes.dex section, just like in the "DEX to grayscale image" approach. These bytes are assigned to the array S . Next, the entropy and proportion of each section is calculated and allocated to the arrays E and P . The padding function needs to be added for the arrays E and P to make them equal to the size of the array S . It should be noted that this is not a zero padding, but a padding where the calculated values are duplicated. Subsequently, the arrays S , E and P , which represent the components of classes.dex, Discrete Entropy and Proportion are concatenated into the three-dimensional array V . Furthermore, the height H and width W values are calculated using the dimensions of the array V . The next step is to use zero padding to ensure the correct shape of the resulting image and reshape the array V itself according to the calculated height H and width W . The final stage of this method is to save the array V as an RGB image in the selected directory. This approach is repeated for all Android applications in the created dataset.

The output of the proposed approach is a set of RGB images, that can be further used as input to the machine learning model. Two randomly selected samples from clean and malicious applications that were processed by the methodology are shown in Figure 24.

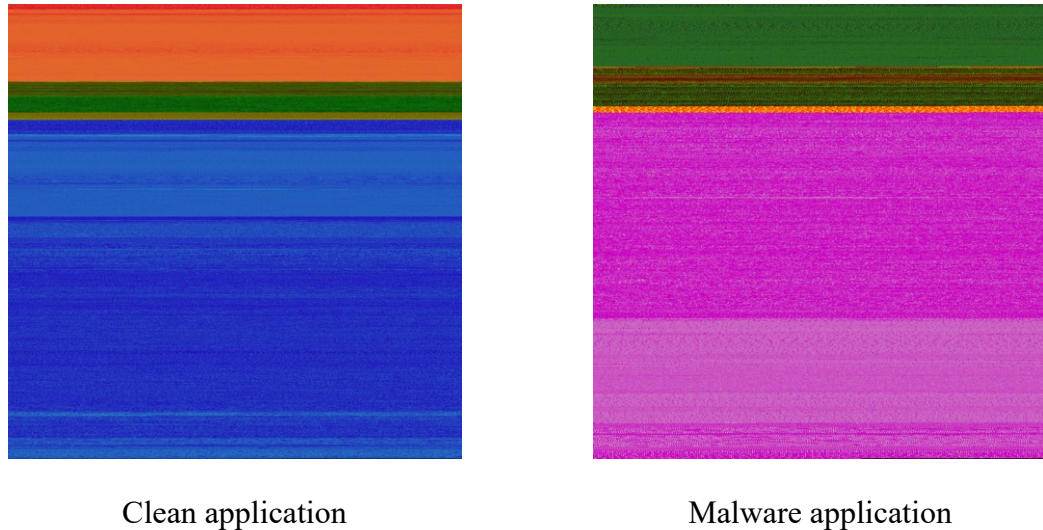


Figure 24. Samples of RGB images for "DEX to RGB image" approach.

7.4 Extraction of AndroidManifest.xml properties

All previous approaches viewed the Android application as an image that can be further processed using computer vision methods. Therefore, one more complementary methodology is presented which deals with the AndroidManifest.xml file that every Android application must contain. To understand the purpose of the AndroidManifest.xml file and its components (for further information see the chapter "*ANDROID OPERATING SYSTEM*"). One of the scientific papers dealing with similar issue is "*Machine Learning for Android Malware Detection Using Permission and API Calls* [12]", which extracts permissions and API calls for further use. The approach taken in this chapter extracts from AndroidManifest.xml the permissions that the application requires (<uses-permission>, <uses-permission-sdk-23>), activities, and features (<uses-feature>). These extracted properties are then compared with the "overall" database to produce a binary sequence, that validates whether the property was present in the application or not. This approach will be used in parallel with one method of perceiving an Android application as an image (the approach selection is discussed in the following chapter) to achieve a more efficient classification. The basic principle of this approach with corresponding the main steps is presented in Figure 25.

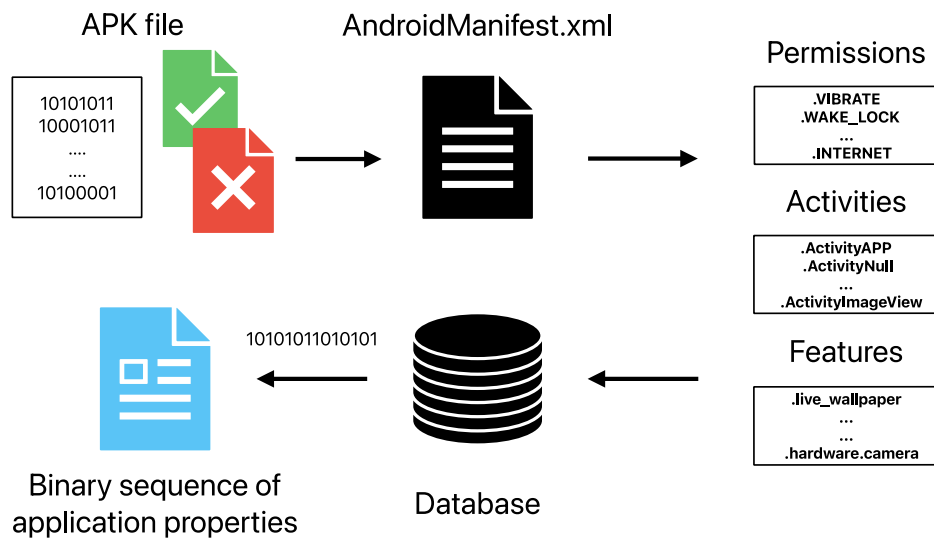


Figure 25. simplified methodology of extracting properties of AndroidManifest.xml.

The detailed description of the steps in this approach showing the procedure in the form of pseudocode is presented in Table 12.

Table 12. Pseudo code that represents the extraction of AndroidManifest.xml properties.

Algorithm 4

PropertiesVector = Array()

Database = LoadPropertiesDatabase()

For *Apk* in *ApkDataset*:

 Open *AndroidManifest* in *Apk*:

PermissionsVector = ExtractAllPermissions(*AndroidManifest*)

ActivitiesVector = ExtractAllActivities(*AndroidManifest*)

FeaturesVector = ExtractAllFeatures(*AndroidManifest*)

PropertiesVector = CreateBinarySequenceOfProperties(
 Database, *PermissionsVector*, *ActivitiesVector*, *FeaturesVector*)

 Save*PropertiesVectorToExcel*(*PropertiesVector*)

Initially, the database needs to be loaded with all the properties contained. Further, after loading the iterated application, the AndroidManifest.xml file is accessed and all

permissions, activities and features are extracted and assigned to the vectors P , A and F . The next step is to compare the properties obtained from the iterated application (vectors P , A and F) against the loaded database which results in a vector V or a sequence of zeros and ones that shows whether the application contained the given property or not. The last step is to write the vector V into Microsoft® Excel® which after preprocessing the whole dataset by this methodology will be utilized as input to the proposed machine learning methods.

To conclude this chapter, Table 13 is established to provide a summary of all the Android application preprocessing approaches that have been presented.

Table 13. Summary of Android application preprocessing approaches.

Approach	Output
APK to greyscale image	Set of greyscale images
DEX to greyscale image	Set of greyscale images
DEX to RGB image	Set of RGB images
AndroidManifest.xml properties	Excel of AndroidManifest.xml properties

8 SELECTION OF EXTRACTED FEATURES FOR INPUT TO MACHINE LEARNING METHODS.

In chapter "*FEATURE EXTRACTION FROM AN ANDROID APPLICATION WITH DETECTION POTENTIAL*", three approaches to visualizing an application as an image were presented. The task of this chapter is to evaluate, which of these approaches exhibit the strongest detection potential. The evaluation process is based on three trivial machine learning models namely Multi-Layer Perceptron, Support Vector Machine (SVM) and Decision Tree Classifier (DTC). Although the chosen methods are not capable of high-quality classification of this complex problem their purpose is to compare and evaluate, which of the approaches is the most appropriate. Figure 26 is designed to describe the methodology, that will be adopted for this process.

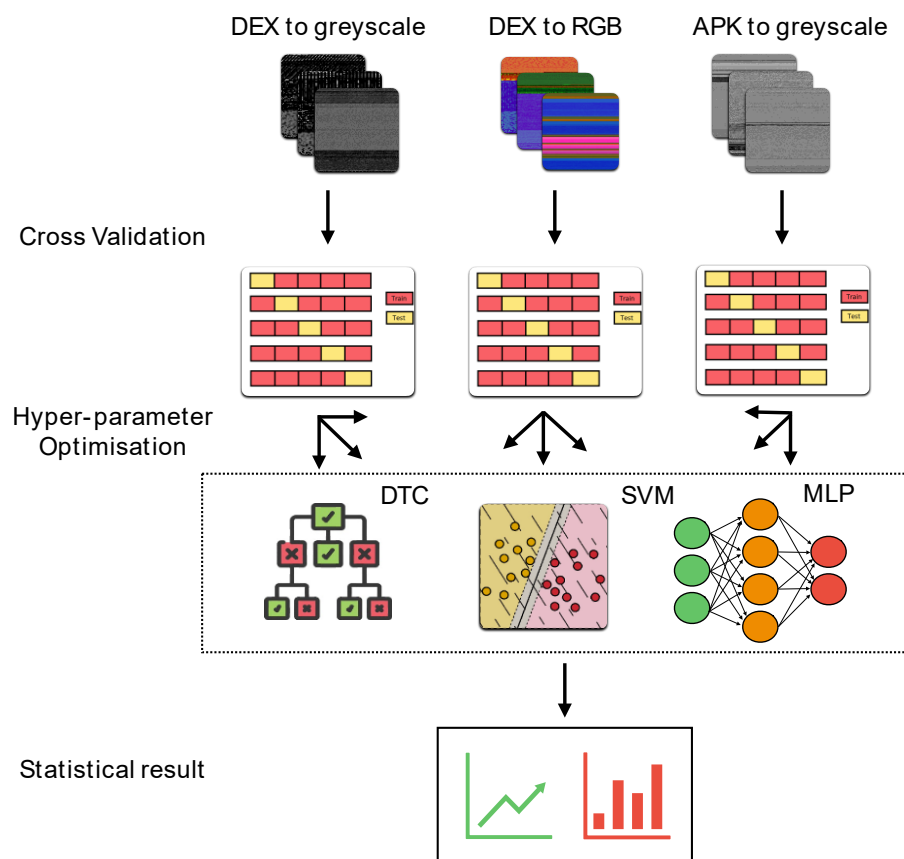


Figure 26. Feature selection process [69, 70].

In Figure 26 it is possible to observe three sets of images representing different approach. Each set of images is then divided using a cross-validation with five folds. K-fold cross-validation is a frequently used method in data science that divides data into K groups of equal size. These groups are called folds. For each training process, the prediction function

uses $(K - 1)$ folds and the remaining one-fold is then used for testing [69]. The partitioned data for each approach are then passed to the selected machine learning methods (DTC, SVM, MLP), where hyper-optimization of model parameters is applied. The hyper-parameter optimization process uses a gradient boosted regression tree method (gbrt), which is improved by sequentially evaluating the expensive function at the next best point. The parameters of the individual models that are subjected to optimization and presented in Table 14

Table 14. Hyper-parameter optimization grid for feature testing.

Label	Limitation
MLP	
Size of hidden layer, S_{hl}	$S_{hl} = \langle 1, 20 \rangle, S_{hl} \in \mathbb{N}$
Learning rate, l_r	$l_r = \langle 0.0001, 0.5 \rangle, l_r \in \mathbb{R}$
Activation function, $F(a)$	$F(a) = \{tanh, relu, logistic\}$
Solver, s_t	$s_t = \{adam, lbfgs, sgd\}$
SVM	
Regularization parameter, R_p	$R_p = \langle 1.0, 6.0 \rangle, R_p \in \mathbb{R}$
Kernel coefficient, K_c	$K_c = \langle 0.5, 7.0 \rangle, K_c \in \mathbb{R}$
Kernel type, K_t	$K_t = \{poly, rbf, sigmoid\}$
DTC	
The minimum number of samples required to split an internal node, SS_{min}	$SS_{min} = \langle 2, 10 \rangle, SS_{min} \in \mathbb{N}$
The minimum number of samples required to be at a leaf node, SL_{min}	$SL_{min} = \langle 1, 7 \rangle, SL_{min} \in \mathbb{N}$
Criterion, C	$C = \{gini, entropy\}$
Splitter, s	$s = \{best, random\}$

The optimization cycle is set to 31 iterations, in other words, 31 runs of the optimization function "gbrt" are reserved for each approach, which implements 50 different settings for

the MLP, SVM and DTC models. The output of this optimization cycle is the mean value of the results from the MPL, SVM and DTC models, respectively. Obtained results, thus reflect the detection potential of the approach. Moreover, these averages can be then compared in the following step using a statistical test to evaluate, which approach yields the best results. Results of the optimization process for all approaches using violin plots are shown in Figure 27.

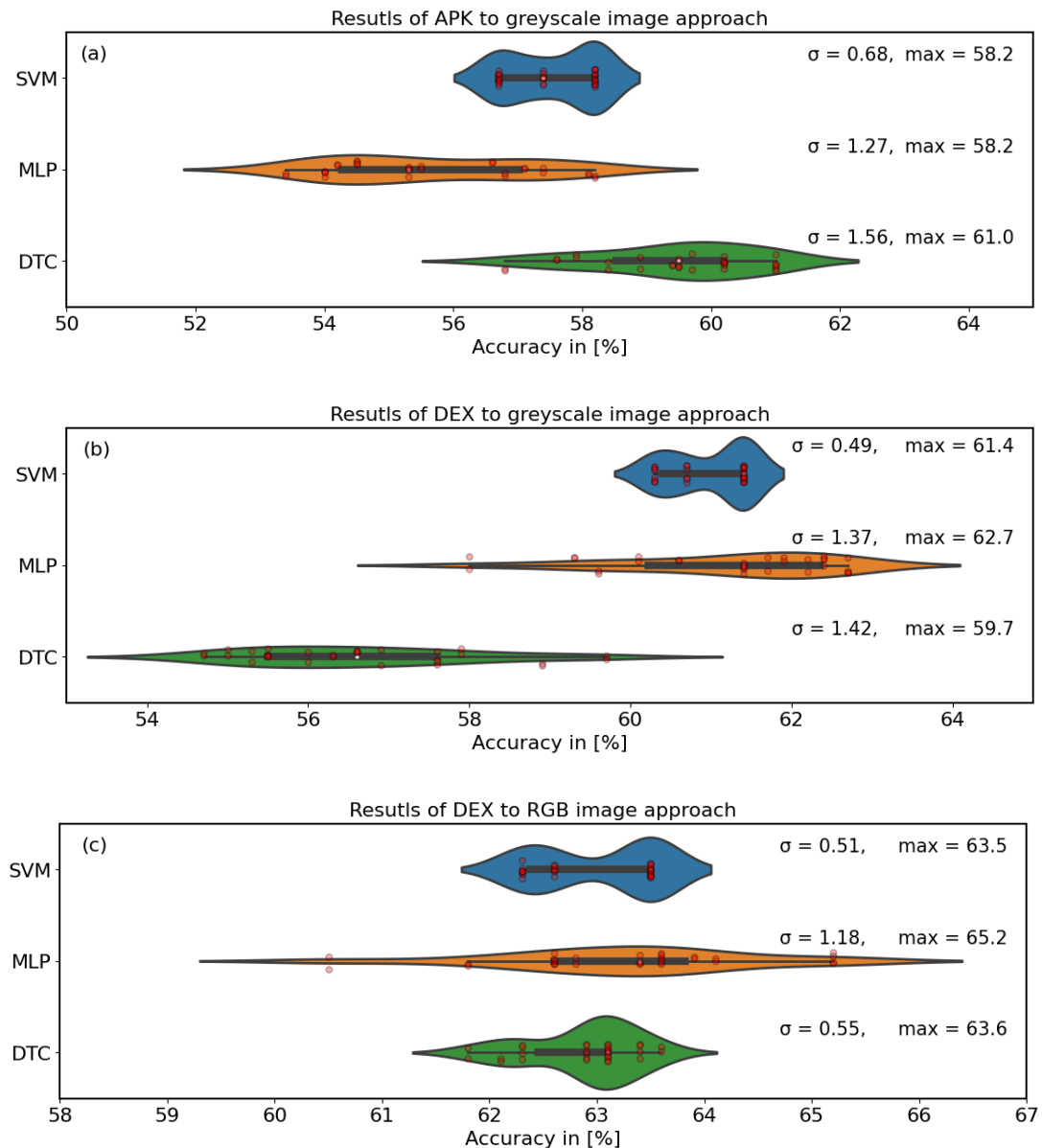


Figure 27. Display of optimization results for testing approaches.

Furthermore, the results obtained using different approaches (Figure 27) are depicted using violin plots that show the distribution of the data, their median and range. In addition, the best value achieved by the model and the standard deviation are added. The graph is divided into three parts (27a, 27b, 27c) where each part shows the results of one approach.

The results collected by "*DEX to RGB image approach*" indicates the strongest detection potential (Figure 27c). In order to declare that this methodology can be considered as a dominant, it is common practice not only to rely on graphical representations but also to perform a correct statistical test. For this purpose, the ANOVA test was chosen. Such a test is commonly used to determine whether there is a statistically significant difference between two or more categorical groups by testing the differences of means using variance [71]. Therefore, the input to the ANOVA test would be the mean values (results from the MLP, SVM and DTC models) from 31 optimization runs for each approach. The result of the test is $p_{val} = 7.8 * 10^{-58}$ indicating a significant difference in means between groups. The next step is to use Tukey's test which will directly show which groups are "different". Table 15 displays the results of the Tukey test.

Table 15. Results of Tukey test.

Group-1	Group-2	Reject
APK to greyscale image	Dex to greyscale image	True
APK to greyscale image	Dex to rgb image	True
Dex to greyscale image	Dex to rgb image	True
Significance level = 0.01		

As shown in Table 15, the selected groups differ from each other, which only proves the hypothesis obtained after graphical representation of the data. According to this validation, it is possible to say that "APK to greyscale image" approach has the weakest detection potential and "*DEX to RGB image*" approach the strongest. For this reason, in the upcoming chapters entire classification will be based on the "*DEX to RGB image*" approach together with the auxiliary "Extraction of AndroidManifest.xml properties" method. In conclusion, it should be noted that the selected testing has some limitations. For example, the optimized parameters of machine learning models are limited, so it is possible that outside these intervals a solution of higher quality can be obtained. Furthermore, 31 optimization runs were performed for each approach and hence it is possible that if the function had more iterations, it would be able to find a more appropriate combination of parameters. However, more extensive testing would have required more powerful computational resources.

9 RESEARCHING AND UTILIZING NEURAL NETWORKS ON THE PROBLEM OF MALICIOUS APPLICATION DETECTION

The previous chapters dealt with the partitioning of the obtained dataset, extraction of features with detection potential and their selection. At this phase, when the data pre-processing phase is complete, it is possible to move on to the actual development of the main machine learning model for Android application classification. The RGB images produced by approach "*DEX to RGB image*" together with the complementary methodology "Extraction of AndroidManifest.xml properties" will be used as input to the model. The splitting of the dataset is performed using the process described in chapter "*DATA GATHERING*", which means that three sets will be used for training and testing, namely the training set, the testing set, and the validation set. Currently, convolutional neural networks are considered one of the most efficient methods for data processing in the subfield of computer vision. With this consideration, also this chapter will primarily focus on the convolutional neural network along with the multilayer feedforward network for auxiliary features. For a more detailed description of neural networks, see "*ARTIFICIAL NEURAL NETWORKS*". The three most commonly used machine learning frameworks in the Python programming language are Pytorch [72], Tensorflow [73] and Scikit-learn [74]. Each of these frameworks presents a number of advantages and is slightly different from the other. Scikit-learn is considered to be very user-friendly and contains a number of built-in models that are relatively easy to use. Therefore, this framework is usually recommended for people new to machine learning. This brings out its biggest drawback and that is due to the fact that the loaded models are a package and changing their architecture and parameters is sometimes very challenging. The TensorFlow framework, although more complicated to learn, is still considered as a user-friendly, mainly because it does many complex operations like convolutional arithmetic automatically, which in the end makes the work much easier. However, compared to Scikit-learn it presents a much higher degree of freedom which gives room to create more complex models. The Pytorch framework gives the user the greatest "freedom" in modifying models and their important elements. For this reason, it is possible to create specialized models in this framework for unique problems such as Android malware detection. Of course, this "freedom" comes with a significant disadvantage, which is the difficulty of learning this framework. Due to my personal experience in all of the libraries described, Pytorch is used in the presented thesis primarily due to its flexibility. Here it should be noted that these are not facts but personal experiences.

9.1 Architecture of the proposed model

Because of the uniqueness of the problem, it is necessary to choose the correct sequence of components to design efficient model for Android malware classification. The proposed model will be further referred to as MD-NNM (Malware Detection-Neural Network model). The first issue arises that most image datasets used in machine learning are normalized to relatively low-resolution images. For example, famous datasets like CIFAR-10 with image sizes of 32×32 pixels or MIT with 128×128 pixels. While in the problem of Android malware detection, such compression at the data pre-processing stage would very likely result in the dataset being unusable. This leads to a dataset of images with dimensions of for example 2048×2048 pixels depending on the size of the applications. Therefore, known convolutional network architectures such as ResNet are applicable but would require high computational requirements. Thus, the situation requires the creation of a model that is custom-made. Figure 28 presents the proposed architecture of the MD-NNM, which will be subsequently used for classification of the Android applications dataset.

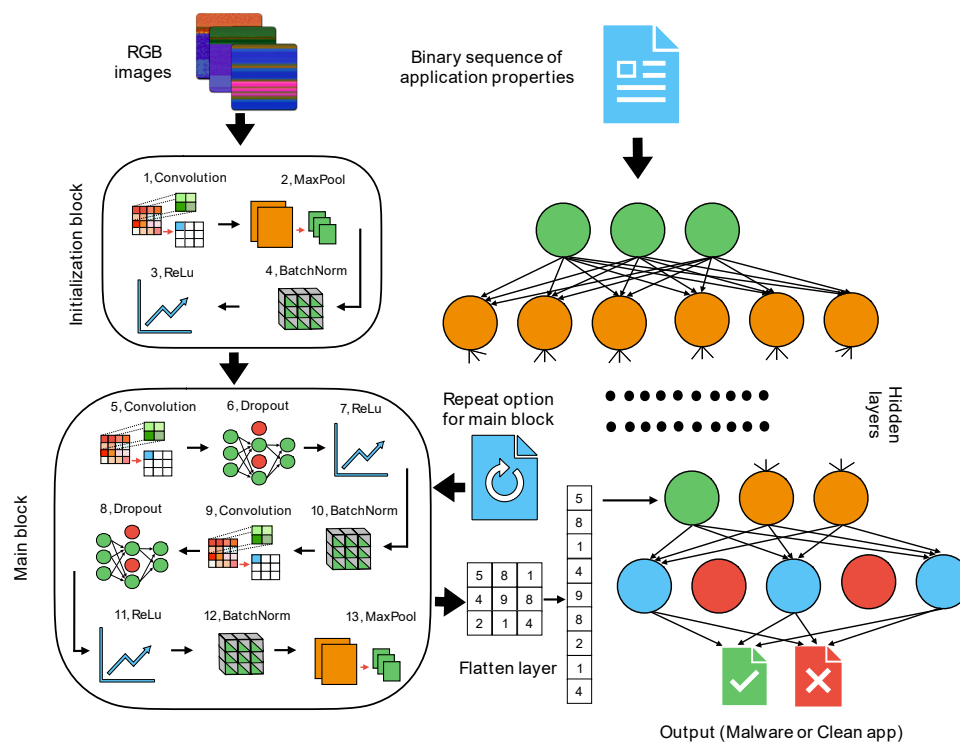


Figure 28. Architecture of MD-NNM.

The MD-NNM architecture can be divided into two main sections namely convolutional neural network and multilayer feedforward network, where their components will

be described and explained in sequence. A set of RGB images is used as input to the part using the convolutional neural network. RGB images in the form of a matrix with the shape $height \times width \times number\ of\ channels$ are fed into the initialization block where the convolutional layer is applied with the following parameters: $kernel\ size = 7 \times 7$, $padding\ size = 1$, $input\ matrix\ depth = 3$ (since these are RGB images), $stride = 3$ and $number\ of\ output\ channels = 64$. It should be noted that certain parameters such as $kernel\ size$ or $padding\ size$ are not fixed and can be used as input to the hyper-optimization of parameters. A given kind of convolutional layer with a large kernel size relative to the other convolutional layers found in this model is often used in high-resolution image problems. The max pooling operation is then applied to the data, which selects the maximum element from the given map feature region that is covered by the filter. Hence, the output of this layer is a feature map containing the most prominent features. This is followed by the batch normalization layer, which allows each layer of the network to do the learning more independently. This feature is used to normalize the output from the previous layer. Batch normalization process generally increases the efficiency of learning and is sometimes used for regularization, which prevents overfitting of the model. The last part of the initialization block is the ReLu function preventing the exponential growth of computational operations, which generally increases the learning speed of large networks. Another advantage of the ReLu function is, that it partially prevents the vanishing gradient phenomenon which is quite common in other functions. The processed data then enters the main block which consists of a convolutional layer with the following parameters: $input\ channel\ size = 64$ (which is equal to the output size from the previous convolutional layer), $kernel\ size = 3 \times 3$, $padding\ size = 3$ and $stride = 2$. As with the first convolutional layer, the parameter values are open to hyper-optimization. Subsequently, the data are fed to the dropout layer which serves as a mask that nullifies the contribution of some elements to the next layer. The dropout layer is an important tool that partially prevents network overfitting. The dropout coefficient serves as another feature of hyper optimization, and it will range in the interval $(0.2, 0.9)$. Then the data passes through the ReLu function, a batch normalization layer and another convolution layer where its outcome will be equal to $output\ channel = 128$ in this instance. This is followed by another Dropout layer, a ReLu function, a batch normalization layer and finally a max pool layer. It should be noted that the main block can be repeated several times. In the case of repeating this block, the sequence of operations and their

parameters remain the same except for the size of the *input* and *output channels* which take the values {64,128,256,512 ...}. After that, the processed data are adjusted by a flattening layer to be able to enter the final fully connected layer. In parallel to this, data extracted from AndroidManifest.xml are inserted into the section using the forward network, which consists of fully connected layers and ReLu functions. Data from both sections are then joined together in the last fully connected layer where a dropout layer is added before evaluating whether the application is clean or malicious.

9.2 Key features of MD-NNM

The next important step in understanding MD-NNM is the specification of the remaining elements. Cross entropy loss is used as a criterion to stop the process, which acts as a metric to determine how well the chosen machine learning model classifies the data. The loss is measured as a number between zero and one where zero indicates the ideal outcome. In general, Cross entropy loss can be expressed using equation 5.

$$L_{CE} = - \sum_{i=1}^n t_i \log(p_i), \quad \text{for } n \text{ classes} \quad (5)$$

where t_i is the truth label and p_i is the SoftMax probability for the i -th class. When training a machine learning model, its internal parameters need to be modified at each new epoch. An optimizer is a function that modifies attributes in this case of MD-NNM such as weights or learning rate. Nowadays, there are several sophisticated optimizers like Stochastic Gradient Descent (SGD), Adaptive Moment Estimation (ADAM), Limited-memory Broyden–Fletcher–Goldfarb–Shanno (LBFGS) and therefore this will be one of the variables for the hyper-optimization of MD-NNM parameters. Another important feature is the implementation of the "lightning epoch" this algorithm compares the results of the loss function between the training and validation sets at each epoch and terminates learning when significant overfitting starts to occur. Furthermore, an algorithm using the cyclic learning rate (CLR) is implemented. In the CLR, the learning rate is varied between a lower and a higher threshold. In other words, the learning rate oscillates between the baseline (minimum) learning rate and the maximum learning rate [75]. Here, it is important to note that this algorithm is only implemented for optimizers that do not have a learning rate modification embedded in them.

9.3 Preview of MD-NNM training

With the architecture and components of MD-NNM described, the next step is to illustrate the learning process on the created dataset. This subchapter is not aimed at achieving the best results (this issue is dealt with in Chapter “*OPTIMIZATION AND RESULTS OF THE OBTAINED MODELS*”), but only to demonstrate how the overall process looks like and what kind of results can be expected. The first step is to select the model parameters. Here the hyper optimization process is ignored for the time being and the parameters will be chosen “randomly”. Table 16 shows selected MD-NNM parameters.

Table 16. Randomly selected MD-NNM parameters.

MD-NNM	
Parameter	Value
Criterion	Cross entropy loss
Optimizer	SGD
Learning rate	$CLR = 0.001$
Dropout coefficient	$D_C = 0.7$
Initial convolution layer	$kernel\ size = 7 \times 7$
	$padding\ size = 1$
	$stride = 3$
Other convolution layers	$kernel\ size = 3 \times 3$
	$padding\ size = 3$
	$stride = 2$
Activation function	ReLU
Repetitions of the main block	$repeat_{count} = 1$

Subsequently, the model with the selected parameters is learned on a training set along with a validation set which serves as an accuracy check during learning. To understand the learning process as a whole, a depiction of its fundamental characteristics is required.

The loss function values for the training and validation set and the development of the learning rate is presented in Figure 29.

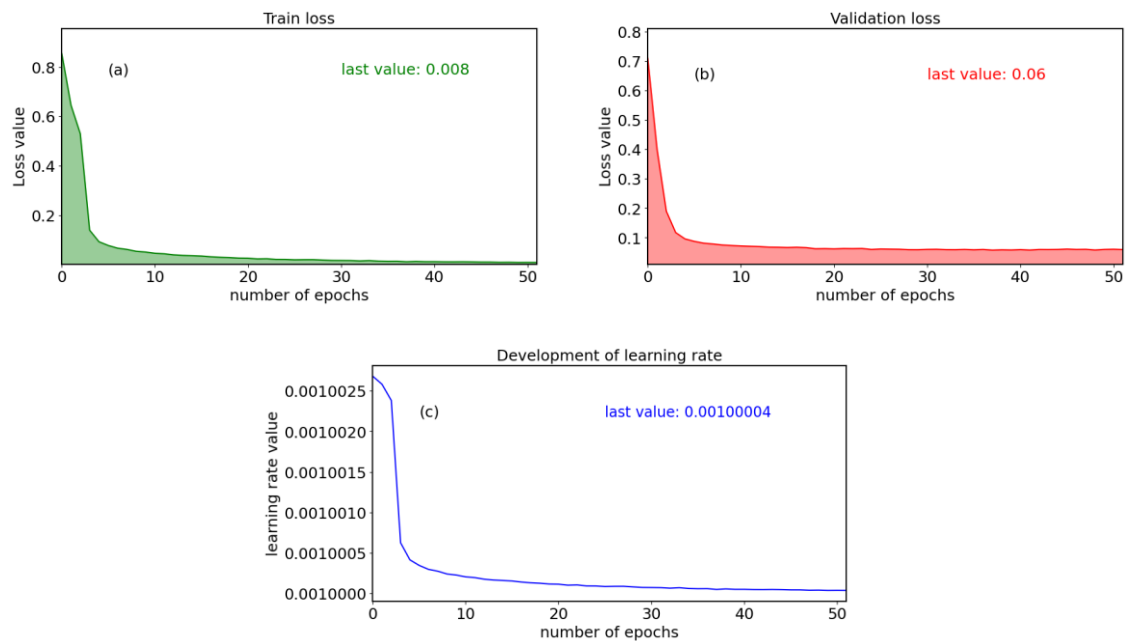


Figure 29. The showcase of learning process for MD-NNM.

Figure (29a, 29b) displays the trend of the loss function for the training and validation set. It can be observed, that there is no significant overfitting in the whole process. This is due to implemented measures such as dropout layers. It should be pointed out, that for different parameter settings the learning process might look diametrically different and that is why safety mechanisms like the lightning epoch are implemented. Figure 29c shows the evolution of the learning rate where it can be seen, that at the beginning there are higher values because the algorithm tries to utilize the neighborhood of the function and towards the end the values are much smaller (exploitation phase). To get an indication of how effective a given model is, Table 17 displays the accuracies of the training, testing, and validation sets as well as showing the confusion matrix.

Table 17. Accuracy and Confusion matrix of test process of MD-NNM.

Type of set	Accuracy [%]	Confusion matrix		
Train set	99.91	Clean	337 (TP)	3 (FP)
Validation set	97.97	Malware	8 (FN)	331 (TN)
Test set	98.53		Clean	Malware

10 UTILIZATION AND EXPLORATION OF NOVEL MACHINE LEARNING METHOD TO ACHIEVE EXPERIMENTAL RESULTS

Despite the dominance of convolutional networks in computer vision, a new approach to image processing is nowadays coming to the fore. This approach is known as a transformer that was originally developed for natural language processing but over time variants such as Vision Transformer [27] were created for image processing. This chapter aims to implement a novel ViT approach in combination with feedforward network on the Android malware detection problem. As in the chapter "*RESEARCHING AND UTILIZING NEURAL NETWORKS ON THE PROBLEM OF MALICIOUS APPLICATION DETECTION*", Android applications that are preprocessed by the approaches "*DEX to RGB image*" and "*Extraction of AndroidManifest.xml properties*" will be fed into this model. To understand how transformers function and how they differ from convolutional neural networks, please see the chapter "TRANSFORMERS" in the theoretical part of the thesis. The data entering the model will be split using the procedure described in the chapter "*DATA GATHERING*", in other words the training, testing and validation set will be used for the learning process and subsequent testing.

10.1 Architecture of the experimental model

The architecture of the model is therefore composed of two main parts ViT, which is used to process the images (in this case RGB images) and a multi-layer feed forward network to process the features extracted from AndroidManifest.xml. The created model will be further referred to as MD-ViTNNM (Malware Detection - Vision Transformer Neural Network Model). The approach of ViT involves breaking down an image into smaller sections called "patches" and representing each patch with a linear embedding. These embeddings are treated as tokens, similar to words in natural language processing tasks. The Transformer model is then trained on image classification using a supervised method. It is generally known that transformers used for image processing work better on larger datasets, which could cause a potential concern in this case (new studies are working on this problem such as Data-efficient image transformers - DeiT) [76]. For this reason, the AndroidManifest.xml properties are processed in parallel using the feedforward network as a complimentary resource. Finally, the processed data will be combined into a final layer from which the classification of whether the application is clean or malicious will be made.

Figure 30 shows the proposed MD-ViTNNM architecture, which will be described and explained.

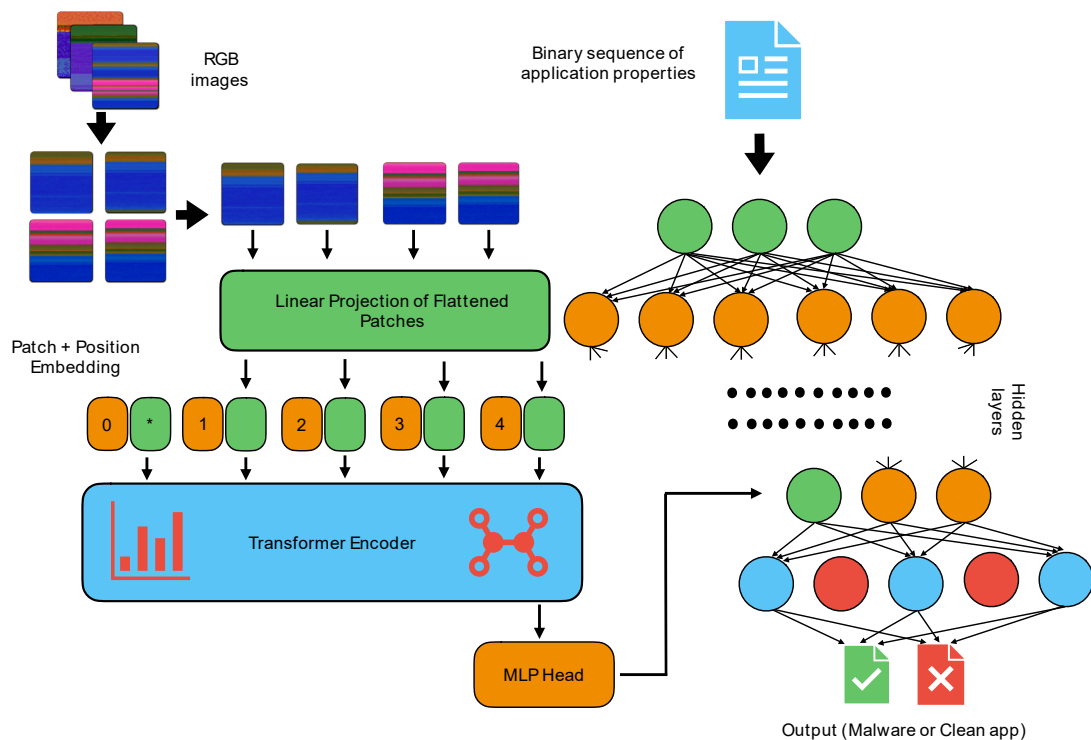


Figure 30. Architecture of MD-ViTNNM.

The input image in the shape of $height \times width \times number\ of\ channels$ (in this case RGB) is first divided into smaller patches, typically of size $S = 16 \times 16\ pixels$, with a $stride = 16$ (size and stride are subjects for hyper optimization of parameters). Each patch is then linearly embedded, meaning it is transformed into a multi-dimensional feature vector. These feature vectors are then treated as tokens. The tokens are then processed by a multi-head self-attention mechanism in the self-attention layers. This mechanism allows the model to attend to different parts of the input, which is crucial step for image processing as it allows the model to focus on different regions of the image. The tokens then go through feed-forward layers, which apply a linear transformation followed by a non-linear activation function (Gaussian Error Linear Units - GELU). This allows the model to learn a non-linear mapping of the input features. The output of the last layer is then passed through a linear layer to obtain a feature vector of the same dimension as the input patch. The feature vectors of all patches are then concatenated and passed through a linear layer. In parallel to this process, data extracted from AndroidManifest.xml are loaded into the section using the forward network, which is composed of fully connected layers and ReLu functions.

Data from both sections are then linked together in the last fully connected layer where a dropout layer is added before evaluating whether the application is clean or malicious. As with MD-NNM, the same key features are used. That is, the cross-entropy loss criterion is used to stop the process, the optimizer is chosen as a variable to the hyper-optimization from the set $\{SGD, ADAM, LBFGS\}$ and auxiliary algorithms such as the lightning epoch and CLR are implemented.

10.2 Preview of MD-ViTNNM training

The next stage in developing a MD-ViTNNM is to show the training process of the model using the dataset that has been created. The aim of this subchapter is not to attain the highest possible performance, but rather to provide an overview of the overall process and the expected outcomes. The first step is to select the model parameters. In this instance, the process of hyperparameter optimization is skipped, and the parameters will be chosen “randomly”. Table 18 illustrates the selected MD-ViTNNM parameters.

Table 18. Randomly selected MD-ViTNNM parameters.

MD-ViTNNM	
Parameter	Value
Criterion	Cross entropy loss
Optimizer	SGD
Learning rate	$CLR = 0.001$
Patch size	$P_s = 64$
Number of blocks	$B_n = 8$
Number of heads	$H_n = 2$
Activation function	ReLu, GELU
Embedding dimensions	$E_d = 768$

After selecting the parameters of MD-ViTNNM, the model is trained using a training set and a validation set is used to monitor the model accuracy during training. To understand the training process as a whole, it is helpful to visualize its key characteristics. Figure 31

illustrates the loss function values for the training and validation set, as well as the changes in the learning rate over time.

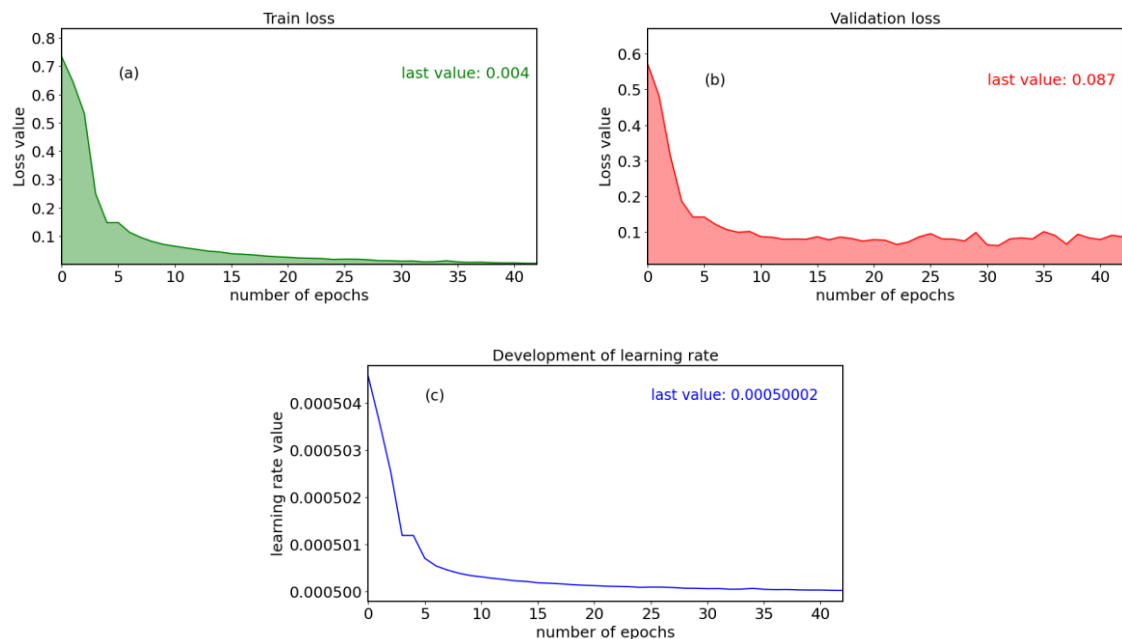


Figure 31. showcase of learning process for MD-ViTNNM.

The loss function values (Figure 31 a, 31 b) show a reasonably robust decrease for the training and validation sets. For the validation set, a slightly fluctuating tendency of the loss function values can be observed in later epochs. This behavior can be considered relatively normal and by adjusting the parameters of the mechanisms to prevent overfitting, a smoothing of the trend can be expected. In the behavior of the learning rate (Figure 31c), it is possible to observe a steep decline in its values at the beginning, followed by a “steady” decline state lasting until the end of the process. The next step is to illustrate the accuracies for the training, validation, and test sets of MD-ViTMM and the confusion matrix depicting the ratio between correctly and incorrectly classified samples (Table 19).

Table 19. Accuracy and Confusion matrix of test process of MD-ViTNNM.

Type of set	Accuracy [%]	Confusion matrix		
Train set	99.97	Clean	335 (TP)	5 (FP)
Validation set	96.71	Malware	7 (FN)	332 (TN)
Test set	97.83		Clean	Malware

11 OPTIMIZATION AND RESULTS OF THE OBTAINED MODELS

The chapters *"RESEARCHING AND UTILIZING NEURAL NETWORKS ON THE PROBLEM OF MALICIOUS APPLICATION DETECTION"* and *"UTILIZATION AND EXPLORATION OF NOVEL MACHINE LEARNING METHOD TO ACHIEVE EXPERIMENTAL RESULTS"* addressed the implementation of machine learning models for malicious application detection. It is now necessary to determine which of the proposed models shows the best results. The evaluation process is based on hyper-optimization of parameters using the Tree-structured Parzen Estimator (TPE) algorithm. In addition to the implemented MD-NNM and MD-ViTNNM, two other models, namely ResNet [77] and DenseNet [78], will be subjected to the optimization process. These are world-renowned convolutional neural networks, and their purpose is to get an idea of how effective they are relative to custom-built models for this issue. Table 20 shows the individual parameters of the models that are subjected to the hyper-optimization process.

Table 20 *Hyper optimization parameters*

Model	Parameter	Value
MD-NNM	Optimizer	$\{SGD, ADAM, LBFGS\}$
	Learning rate	$\langle 0.0005, 0.01 \rangle$
	Dropout coefficient	$\langle 0.1, 0.9 \rangle$
	Initial convolution layer	$kernel\ size = \langle 7 \times 7, 14 \times 14 \rangle$ $padding\ size = \langle 1, 10 \rangle$ $stride = \langle 3, 12 \rangle$
	Other convolution layers	$kernel\ size = \langle 3 \times 3, 7 \times 7 \rangle$
	Repetitions of the main block	$\langle 1, 2 \rangle$
MD-ViTNNM	Optimizer	$\{SGD, ADAM, LBFGS\}$
	Learning rate	$\langle 0.0005, 0.01 \rangle$
	Dropout coefficient	$\langle 0.1, 0.9 \rangle$

	Patch size	$\langle 64, 512 \rangle$
	Number of blocks	$\langle 2, 10 \rangle$
	Number of heads	$\langle 1, 4 \rangle$
ResNet		
	Learning rate	$\langle 0.0005, 0.01 \rangle$
	Number of layers	$\langle (1,1,1,1), (2,2,2,2) \rangle$
DenseNet		
	Learning rate	$\langle 0.0005, 0.01 \rangle$
	Number of layers	$\langle (1,1,1,1), (2,2,2,2) \rangle$

It can be noticed that the hyperparameters of ResNet and DenseNet are considerably reduced compared to the custom models. This is due to the high computational complexity of the networks, which limits their full use. Testing of models was done in Google Colab Pro environment with technical parameters of Graphics processing unit - GPU (K80, T4 and P100) and random-access memory - RAM 32 GB which helps to significantly accelerate the process of training and optimization.

11.1 Optimization Process

Each model was tested on the established dataset with appropriate partitioning into training, validation, and test sets. RGB images and features extracted from AndroidManifest.xml are used as input to MD-NNM and MD-ViTNNM. For the ResNet and DenseNet models, only RGB images serve as input as it would be necessary to interfere with their architecture to add auxiliary inputs. The optimization cycle is set to 10 iterations, in other words there are 10 settings of optimized parameters allocated to each model. The parameter values are determined using the TPE algorithm based on the results of previous iterations. After each iteration, the model is subjected to a test set and the accuracy obtained is stored for subsequent evaluation of the results. The result of the optimization process is therefore a pool of 10 accuracies on the test set for each model. To get a simplified overview of how the optimization process works, its representation is shown in Figure 32.

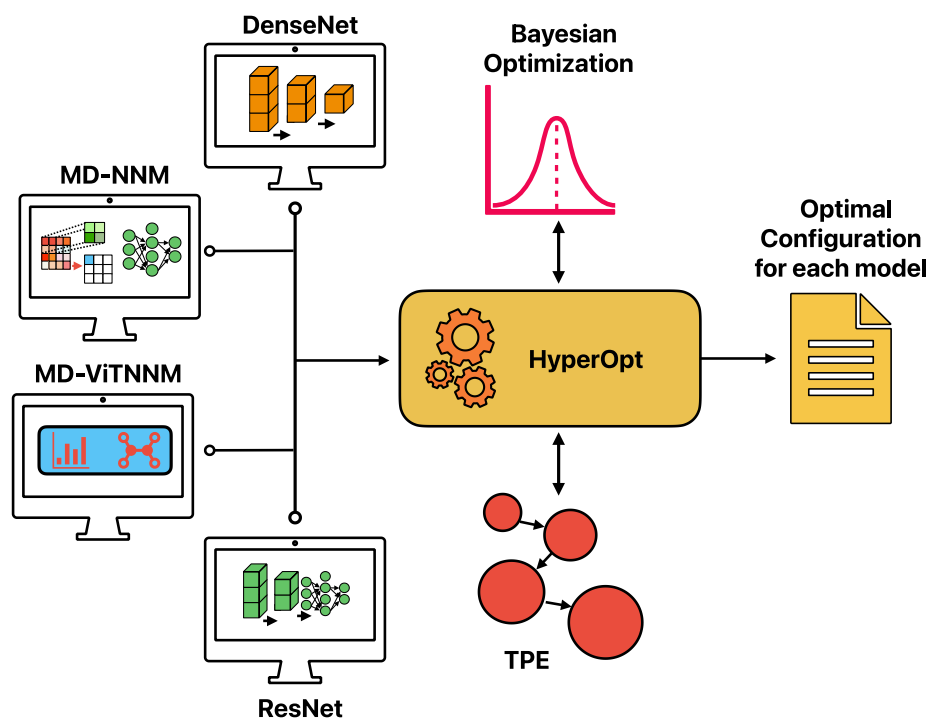
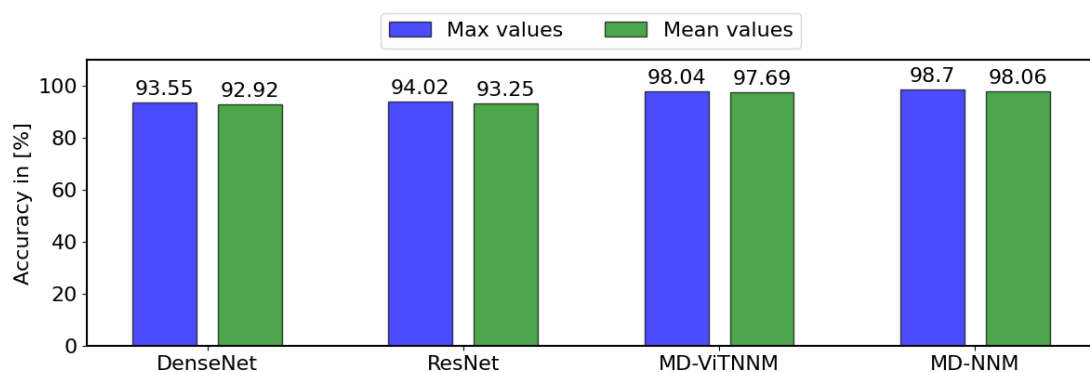
Figure 32 *Optimization process*

Figure 32 shows four selected machine learning models that are sequentially subjected to hyper-optimization of parameters using the TPE algorithm which finds the "optimal" solution in a selected number of iterations.

11.2 Results and discussion

The results of the optimization process are presented in Figure 33. For each model, their best accuracy on the test set is shown along with the average value across all iterations.

Figure 33 *Results of the optimization process*

The custom-made models (MD-ViTNNM, MD-NNM) to which this research has been devoted exhibit significantly better results than the conventional networks. It can be observed that both models exceed the 98% threshold, which can be considered

as an effective solution to the problem of Android malware detection. The DenseNet and ResNet models show accuracies ranging between (93 – 94)% proving that the selected feature extraction method (*DEX to RGB image approach*) indeed exhibits strong detection potential. However, as the results show only one extraction methodology is not sufficient and therefore custom models that also use auxiliary features extracted from AndroidManifest.xml achieve noticeably better results.

Here it should be noted that the testing performed has several limitations. First, the ResNet and DenseNet models had very limited parameters, see Table 20, so testing them due to technical constraints may not have revealed their full protentional. Furthermore, the optimization cycle was set to only 10 iterations so it is likely that for each model there are unknown settings that can be utilized to achieve even better results. The last limitation and the most significant is the size of the dataset. The dataset contains 3391 applications, which is not a statistically insignificant sample, but there are millions of applications on the market. Therefore, it is strongly recommended to increase the number of applications in the dataset in the follow-up research.

Furthermore, due to the high demanding nature of the problem on the computational technology, it is recommended to use the following methods in the subsequent research. Distillation, which uses a teacher model that dramatically increases the speed and in specific cases the quality of learning [79]. Regularization that prevents the model from learning noise in the data but only learns relevant information. Note that data regularization is utilized in this research, but further experimentation is recommended. Data augmentation which generally increases the robustness of the model and its results on the test set. However, with data augmentation, increased care and a good understanding of how the Android application works is required.

12 IMPLEMENTATION OF THE SELECTED MODEL PRODUCING THE BEST RESULTS FOR MALICIOUS APPLICATION DETECTION ON ANDROID PLATFORM.

The previous chapter concluded that the developed models (MD-NNM, MD-ViTNNM) for Android malware detection achieved promising results. Therefore, both models will be implemented for follow-up research in the PT-LAB. So far, all data processing and model testing has been programmed in Google Colab Pro and Visual Studio Code (Jupyter extension) environments which are considered the norm in machine learning circles. Now the trained models along with all auxiliary methodologies will be transferred to the PyCharm environment. The reason for this transition is to consolidate the processed topic into a project format that authorized persons will be able to download and use. In Figure 34 it is possible to see the architecture of the created project.

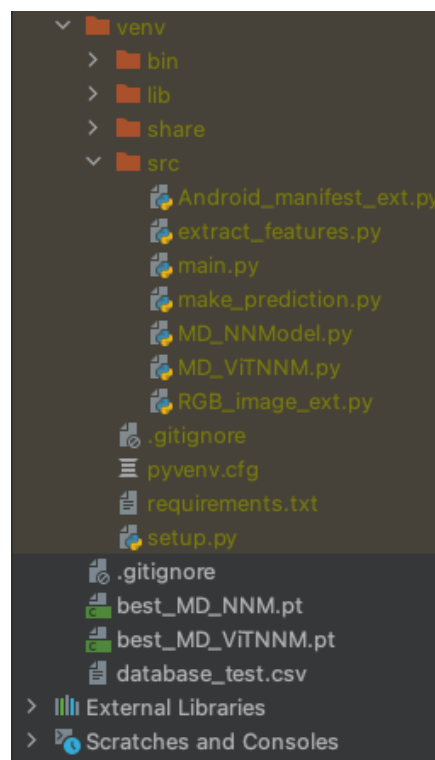


Figure 34 *Project architecture in the PyCharm environment.*

The src folder contains the modules that provide the application logic of the project. The first step is to obtain the properties from the Android application. For this stage, the modules *Android_manifest_ext.py* which extracts permissions (<uses-permission>, <uses-permission-sdk-23>), activities, and features (<uses-feature>) from AndoridManifest.xml and *RGB_image_ext.py* that creates an RGB image from the application dexes are used. To

understand the procedure of extracting the properties, see chapters "*Extraction of AndroidManifest.xml properties*" and "*DEX to RGB image approach*". Subsequently, these methodologies are invoked by the *extract_features.py* module which loads the Android application(s) from the selected directory and pre-processes the extracted features in order to use them as inputs to the developed models. The modules *MD_NNModel.py* and *MD_ViTNNM.py* provide the architecture of the presented models implemented using the Pytorch library. It should be noted that the files *best_MD_NNM.pt* and *best_MD_ViTNNM.pt* contain the learned model parameters obtained in the chapter "*OPTIMIZATION AND RESULTS OF THE OBTAINED MODELS*" using "torch.save()" which is a way of saving and loading models in the Pytorch library [72]. Furthermore, all the preceding actions are combined in the *make_prediction.py* module in which the extracted and pre-processed properties from the Android application are fed into the selected model. The output of this module is a prediction of whether the application is clean or malicious. The last step is calling the *main.py* module which converts the obtained prediction into a graphical environment for user comfort.

12.1 Project functionality demonstration

Now a demonstration of how the project works will be made and described. After starting the project, the graphical interface appears, which can be seen in Figure 35.

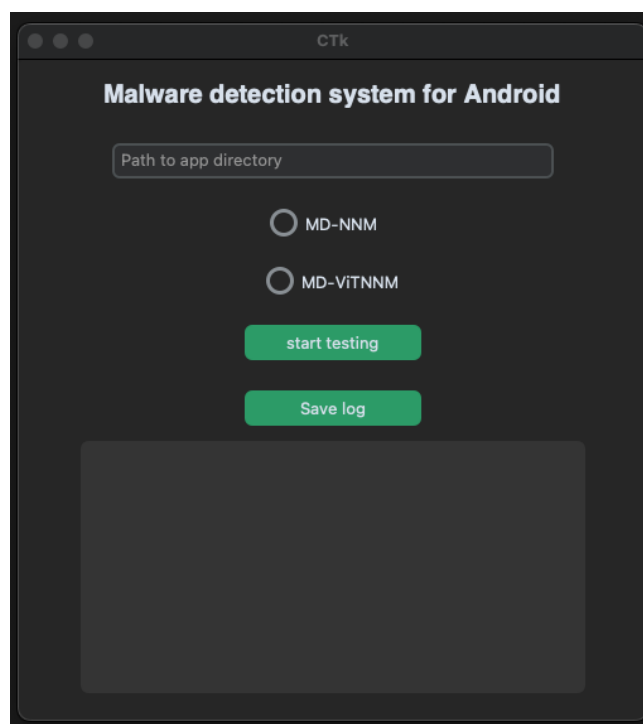


Figure 35 *Graphical interface of the project*

Here it is necessary to enter the path that leads to the folder, where the Android applications that require testing are located. Next, the model needs to be selected. There are two models to choose from (MD-NNM and MD-ViTNNM) to which this thesis has been dedicated. After selecting the model and specifying the path to the applications, it is necessary to click on the *start testing* button which launches the program directly. Figure 36 shows one of the available configurations.

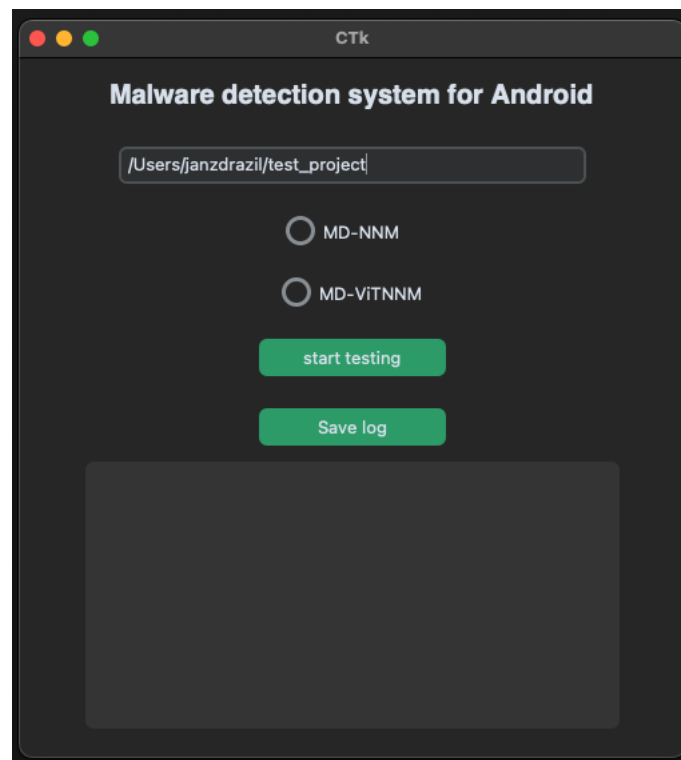


Figure 36 Possible Graphical Interface Settings

When the button *start testing* is activated, all applications (.apk) in the folder will be tested. The time required for testing depends on the number of applications in the folder and their size. During the process, applications that have already been tested will be gradually added to the *text window* in the following format: application name and whether the application is clean or malicious. The outcome of such testing is depicted in Figure 37.

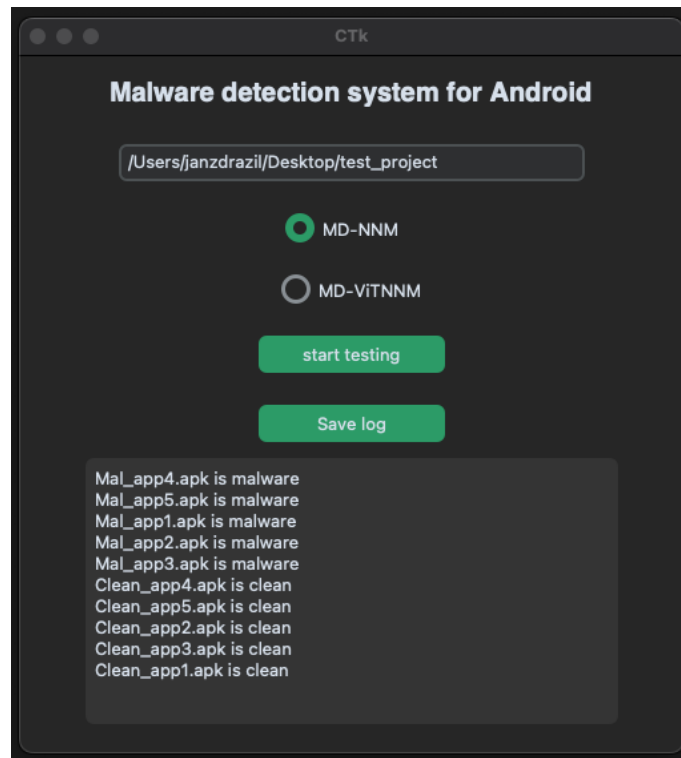


Figure 37 Testing result.

CONCLUSION

At the beginning of the thesis, expert research was conducted addressing the current state of the art of Android malware detection. The aim of this research was to provide a fundamental understanding of the topic and to select the appropriate methods for feature extraction from Android applications and machine learning techniques. Subsequently, the thesis focuses on the overall description of the Android OS architecture and its main components. Further, the basics of ANNs were discussed, starting from the simplest version like perceptron to more advanced such as CNNs. Since one of the aims of the thesis is the implementation of the novel methods, a portion of the theory is devoted to transformers, especially to a modified version called ViT. The conclusion of the theoretical part of the thesis is dedicated to the basics of hyper-optimization of parameters whose application is necessary for tuning machine learning models.

The first task of the practical part of the thesis is focused on data gathering, where data assume the format of android applications and their subsequent division into training, testing and validation sets. With the properly partitioned dataset, four approaches dealing with feature extraction from android application are subsequently presented and described. The approaches are then subjected to testing using simple machine learning methods and hyper-optimization to reveal their detection potential. Subsequently, the thesis introduces a custom-made MD-NNM model that combines the potency of the CNN for image shape features and the feedforward network for auxiliary approach. Following this, an experimental model MD-ViTNNM is presented which utilizes a special variant of transformer called ViT instead of the conventional CNN. To be able to compare the performance of the presented models, the renowned DenseNet and ResNet models were employed in the study, followed by their comparison on the test set using hyper-optimization of parameters. This measurement revealed that the custom-built models (MD-NNM, MD-ViTNNM) performed significantly better than the imported ones. Despite the study exhibiting promising results, the thesis points out limitations of the testing such as restricted processing power and the size of the datasets. For these reasons, it is recommended to re-train the models on a larger dataset before using them in practice. The result of the thesis are models integrated into a project format with a graphical interface that can be used for testing downloaded applications running under Android OS.

BIBLIOGRAPHY

- [1] Android Malware. *Av-test.org* [online]. The Independent IT-Security Institute [cit. 2023-03-27]. Available from: <https://www.av-test.org/en/statistics/malware>
- [2] Threat Insights Report 2020. *Pandasecurity.com* [online]. [cit. 2023-03-27]. Available from: <https://www.pandasecurity.com/en/mediacenter/panda-security/threat-insights-report-2020>
- [3] Testing of free AV. *Comparitech.com* [online]. [cit. 2023-03-27]. Available from: <https://www.comparitech.com>
- [4] TILBORG, Henk C. A. van a Sushil JAJODIA. Encyclopedia of cryptography and security. 2nd ed. New York: Springer, c2011. Springer reference. ISBN 9781441959065.
- [5] Why Signature-Based Detection Struggles to Keep Up with the New Attack Landscape?, 2022. *Cybersecuritynews.com* [online]. [cit. 2023-03-27]. Available from: <https://cybersecuritynews.com/signature-based-detection/>
- [6] SHOAIB AKHTAR, Muhammad a Tao FENG, 2022. Malware Analysis and Detection Using Machine Learning Algorithms. *Symmetry* [online]. [cit. 2023-03-27]. Available from: <https://www.mdpi.com/2073-8994/14/11/2304>
- [7] *IEEE Xplore Digital Library* [online]. [cit. 2023-03-27]. Available from: <https://ieeexplore.ieee.org/Xplore/home.jsp>
- [8] *ScienceDirect* [online]. [cit. 2023-03-27]. Available from: <https://www.sciencedirect.com>
- [9] *SpringerLink* [online]. [cit. 2023-03-27]. Available from: <https://link.springer.com>
- [10] *GoogleScholar* [online]. [cit. 2023-03-27]. Available from: <https://scholar.google.com>
- [11] PAN, Ya, Xiuting GE, Chunrong FANG a Yong FAN. A Systematic Literature Review of Android Malware Detection Using Static Analysis. *IEEE Access* [online]. 2020, 8, 116363-116379 [cit. 2023-03-27]. ISSN 2169-3536. Available from: doi:10.1109/ACCESS.2020.3002842
- [12] PEIRAVIAN, Naser a Xingquan ZHU. Machine Learning for Android Malware Detection Using Permission and API Calls. In: *2013 IEEE 25th International Conference on*

Tools with Artificial Intelligence [online]. IEEE, 2013, 2013, s. 300-305 [cit. 2023-03-27]. ISBN 978-1-4799-2972-6. Available from: doi:10.1109/ICTAI.2013.53

[13] AGRAWAL, Rishab, Vishal SHAH, Sonam CHAVAN, Ganesh GOURSHETE a Nahid SHAIKH. Android Malware Detection Using Machine Learning. In: 2020 International Conference on Emerging Trends in Information Technology and Engineering (ic-ETITE) [online]. IEEE, 2020, 2020, s. 1-4 [cit. 2023-03-27]. ISBN 978-1-7281-4142-8. Available from: doi:10.1109/ic-ETITE47903.2020.491

[14] FAN, Ming, Jun LIU, Xiapu LUO, Kai CHEN, Zhenzhou TIAN, Qinghua ZHENG a Ting LIU. Android Malware Familial Classification and Representative Sample Selection via Frequent Subgraph Analysis. *IEEE Transactions on Information Forensics and Security* [online]. 2018, **13**(8), 1890-1905 [cit. 2023-03-27]. ISSN 1556-6013. Available from: doi:10.1109/TIFS.2018.2806891

[15] DUBOVIKOV, Kirill, 2019. Managing Data Science. Packt Publishing. ISBN 9781838824563.

[16] BANOULA, Mayank. Machine Learning Steps: A Complete Guide!. Simplilearn.com [online]. [cit. 2023-03-27]. Available from: <https://www.simplilearn.com/tutorials/machine-learning-tutorial/machine-learning-steps>

[17] KAPRATWAR, Ankita, 2016. *Static and Dynamic Analysis for Android Malware Detection*. USA. Master's Theses and Graduate Research. San Jose State University.

[18] Malware Analysis: Static vs. Dynamic and 4 Critical Best Practices. [www.aquasec.com](https://www.aquasec.com/cloud-native-academy/cloud-attacks/malware-analysis/) [online]. [cit. 2023-03-27]. Available from: <https://www.aquasec.com/cloud-native-academy/cloud-attacks/malware-analysis/>

[19] SHHADAT, Ihab, Bara' BATAINEH, Amena HAYAJNEH a Ziad A. AL-SHARIF. The Use of Machine Learning Techniques to Advance the Detection and Classification of Unknown Malware. *Procedia Computer Science* [online]. 2020, 170, 917-922 [cit. 2023-03-27]. ISSN 18770509. Available from: doi:10.1016/j.procs.2020.03.110

[20] LIN, Wei-Cheng a Yi-Ren YEH. Efficient Malware Classification by Binary Sequences with One-Dimensional Convolutional Neural Networks. *Mathematics* [online]. 2022, 10(4) [cit. 2023-03-27]. ISSN 2227-7390. Available from: doi:10.3390/math10040608

- [21] YAMASHITA, Rikiya, Mizuho NISHIO a Richard Kinh GIAN DO, 2018. Convolutional neural networks: an overview and application in radiology. Springer [online]. [cit. 2023-05-08]. Available from: <https://insightsimaging.springeropen.com/articles/10.1007/s13244-018-0639-9>
- [22] JUNG, Jaemin, Jongmoo CHOI, Seong-je CHO, Sangchul HAN, Minkyu PARK a Youngsup HWANG. Android malware detection using convolutional neural networks and data section images. In: Proceedings of the 2018 Conference on Research in Adaptive and Convergent Systems [online]. New York, NY, USA: ACM, 2018, 2018-10-09, s. 149-153 [cit. 2023-03-27]. ISBN 9781450358859. Available from: doi:10.1145/3264746.3264780
- [23] FANG, Yong, Yangchen GAO, Fan JING a Lei ZHANG. Android Malware Familial Classification Based on DEX File Section Features. *IEEE Access* [online]. 2020, **8**, 10614-10627 [cit. 2023-03-27]. ISSN 2169-3536. Available from: doi:10.1109/ACCESS.2020.2965646
- [24] WU, Dong-Jie, Ching-Hao MAO, Te-En WEI, Hahn-Ming LEE a Kuo-Ping WU. DroidMat: Android Malware Detection through Manifest and API Calls Tracing. In: *2012 Seventh Asia Joint Conference on Information Security* [online]. IEEE, 2012, 2012, s. 62-69 [cit. 2023-03-27]. ISBN 978-1-4673-2261-4. Available from: doi:10.1109/AsiaJCIS.2012.18
- [25] GU, Jiuxiang, Zhenhua WANG, Jason KUEN, et al. Recent advances in convolutional neural networks. *Pattern Recognition* [online]. 2018, **77**, 354-377 [cit. 2023-03-27]. ISSN 00313203. Available from: doi:10.1016/j.patcog.2017.10.013
- [26] VASWANI, Ashish, Noam SHAZEER a Niki PARMAR, 2017. Attention Is All You Need [online]. [cit. 2023-03-27]. Available from: doi:10.48550/arXiv.1706.03762
- [27] DOSOVITSKIY, Alexey, Lucas BEYER a Alexander KOLESNIKOV, 2021. *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale* [online]. [cit. 2023-03-27]. Available from: doi:10.48550/arXiv.2010.11929
- [28] The Rise of Android: Why is Android Successful?, 2023. www.Bankmycell.com [online]. [cit. 2023-05-08]. Available from: <https://www.bankmycell.com/blog/how-many-android-users-are-there>

- [29] Introduction to Android Development, 2022. *Www.geeksforgeeks.org* [online]. [cit. 2023-03-27]. Available from: <https://www.geeksforgeeks.org/introduction-to-android-development/>
- [30] Mobile operating systems' market share worldwide from 1st quarter 2009 to 4th quarter 2022, 2023. *Www.statista.com* [online]. [cit. 2023-03-27]. Available from: <https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/>
- [31] What is Android?. *Www.tutorialspoint.com* [online]. [cit. 2023-03-27]. Available from: https://www.tutorialspoint.com/android/android_overview.htm
- [32] MEIKE, G. Blake a Lawrence SCHIEFER. *Inside the android OS: building, customizing, managing, and operating android system services*. 1. Hoboken: Pearson Education, 2021. ISBN 978-0134096346.
- [33] 5 Reasons Why Android Is So Much More Popular, 2019. *Www.makeuseof.com* [online]. [cit. 2023-05-08]. Available from: <https://www.makeuseof.com/tag/android-more-popular-iphone/>
- [34] Android Operating System. *Www.javatpoint.com* [online]. [cit. 2023-03-27]. Available from: <https://www.javatpoint.com/android-operating-system>
- [35] KOTZIAS, Platon, Juan CABALLERO a Leyla BILGE, 2020. *How Did That Get In My Phone? Unwanted App Distribution on Android Devices* [online]. [cit. 2023-03-27]. Available from: doi:arXiv:2010.10088
- [36] Platform Architecture. *Developer.android.com* [online]. [cit. 2023-03-27]. Available from: <https://developer.android.com/guide/platform>
- [37] Android Architecture, 2021. *Developer.android.com* [online]. [cit. 2023-03-27]. Available from: <https://www.geeksforgeeks.org/android-architecture/>
- [38] Hardware Abstraction Layer (HAL) overview. *Source.android.com* [online]. [cit. 2023-05-08]. Available from: <https://source.android.com/docs/core/architecture/hal>
- [39] Android Architecture. *Www.javatpoint.com* [online]. [cit. 2023-03-27]. Available from: <https://www.javatpoint.com/android-software-stack>

- [40] Android Runtime (ART) and Dalvik. *Source.android.com* [online]. [cit. 2023-03-27]. Available from: <https://source.android.com/docs/core/runtime>
- [41] Android Dalvik Virtual Machine. *Techvidvan.com* [online]. [cit. 2023-03-27]. Available from: <https://techvidvan.com/tutorials/android-dalvik-virtual-machine/>
- [42] Working of Dalvik Virtual Machine in Android. *Data-flair.training* [online]. [cit. 2023-03-27]. Available from: <https://data-flair.training/blogs/android-dalvik-virtual-machine/>
- [43] Difference Between Dalvik and ART in Android. *Www.geeksforgeeks.org* [online]. [cit. 2023-03-27]. Available from: <https://www.geeksforgeeks.org/difference-between-dalvik-and-art-in-android/>
- [44] Get started with the NDK. *Developer.android.com* [online]. [cit. 2023-05-08]. Available from: <https://developer.android.com/ndk/guides>
- [45] Using a Native C Library in Android. *Alvarogarcia7.github.io* [online]. [cit. 2023-05-08]. Available from: <https://alvarogarcia7.github.io/articles/2021/05/02/native-c-library-android/>
- [46] Understanding Android-IV (Native C/C++ Libraries, Java API Framework & System Apps), 2021. *Aasish-mithra.medium.com* [online]. [cit. 2023-03-28]. Available from: <https://aasish-mithra.medium.com/understanding-android-iv-native-c-c-libraries-java-api-framework-system-apps-2bdc658d1fb6>
- [47] Application Fundamentals. *Developer.android.com* [online]. [cit. 2023-03-27]. Available from: <https://developer.android.com/guide/components/fundamentals>
- [48] The activity lifecycle. *Developer.android.com* [online]. [cit. 2023-05-08]. Available from: <https://developer.android.com/guide/components/activities/activity-lifecycle>
- [49] Services overview. *Developer.android.com* [online]. [cit. 2023-05-08]. Available from: <https://developer.android.com/guide/components/services>
- [50] Android - Application Components. *Www.tutorialspoint.com* [online]. [cit. 2023-03-27]. Available from: https://www.tutorialspoint.com/android/android_application_components.htm

- [51] Components of an Android Application, 2021. *Www.geeksforgeeks.org* [online]. [cit. 2023-03-28]. Available from: <https://www.geeksforgeeks.org/components-android-application/>
- [52] Content provider basics. *Developer.android.com* [online]. [cit. 2023-05-08]. Available from: <https://developer.android.com/guide/topics/providers/content-provider-basics>
- [53] Android - Application Components Structure of an Android App Binary (.apk), 2022. *Www.appdome.com* [online]. [cit. 2023-03-27]. Available from: <https://www.appdome.com/how-to/appsec-release-orchestration/appdome-basics/structure-of-an-android-app-binary-apk/>
- [54] App Manifest Overview. *Developer.android.com* [online]. [cit. 2023-03-28]. Available from: <https://developer.android.com/guide/topics/manifest/manifest-intro>
- [55] APK-file. *Ajinasokan.com* [online]. [cit. 2023-03-28]. Available from: <https://ajinasokan.com/posts/apk-file-postmortem/#meta-inf>
- [56] App resources overview. *Developer.android.com* [online]. [cit. 2023-05-08]. Available from: <https://developer.android.com/guide/topics/resources/providing-resources>
- [57] Enable multidex for apps with over 64K methods. *Developer.android.com* [online]. [cit. 2023-03-28]. Available from: <https://developer.android.com/studio/build/multidex>
- [58] DU, Ke-Lin a M. N. S. SWAMY. *Neural Networks and Statistical Learning* [online]. London: Springer London, 2014 [cit. 2023-03-27]. ISBN 978-1-4471-5570-6. Available from: doi:10.1007/978-1-4471-5571-3
- [59] AGGARWAL, Charu C. *Neural Networks and Deep Learning* [online]. Cham: Springer International Publishing, 2018 [cit. 2023-03-27]. ISBN 978-3-319-94462-3. Available from: doi:10.1007/978-3-319-94463-0
- [60] TATSAT, Hariom, Sahil PURI a Brad LOOKABAUGH, 2020. *Machine Learning and Data Science Blueprints for Finance*. O'Reilly Media. ISBN 9781492073000.
- [61] MOHRI, Mehryar, Afshin ROSTAMIZADEH a Ameet TALWALKAR, 2018. *Foundations of Machine Learning*. MIT Press. ISBN 9780262039406.

- [62] ALOM, Md Zahangir, Tarek M. TAHA, Chris YAKOPCIC, et al. A State-of-the-Art Survey on Deep Learning Theory and Architectures. *Electronics* [online]. 2019, 8(3) [cit. 2023-03-27]. ISSN 2079-9292. Available from: doi:10.3390/electronics8030292
- [63] AGRAWAL, Tanay, 2020. *Hyperparameter Optimization in Machine Learning*. Apress. ISBN 9781484265789.
- [64] OWEN, Louis Owen, 2022. *Hyperparameter Tuning with Python*. Packt Publishing. ISBN 9781803241944.
- [65] Hyperparameters Optimization, 2019. *Towardsdatascience.com* [online]. [cit. 2023-03-27]. Available from: <https://towardsdatascience.com/hyperparameters-optimization-526348bb8e2d>
- [66] Hyperparameter Optimization Techniques to Improve Your Machine Learning Model's Performance, 2020. *Www.freecodecamp.org* [online]. [cit. 2023-03-27]. Available from: <https://www.freecodecamp.org/news/hyperparameter-optimization-techniques-machine-learning/>
- [67] CICMalDroid 2020, 2020. *Www.unb.ca* [online]. [cit. 2023-03-27]. Available from: <https://www.unb.ca/cic/datasets/maldroid-2020.html>
- [68] PTLab. *Ptlab.fai.utb.cz* [online]. [cit. 2023-03-27]. Available from: <https://ptlab.fai.utb.cz>
- [69] Cross Validation. *Aiaspirant.com* [online]. [cit. 2023-03-27]. Available from: <https://aiaspirant.com/cross-validation/>
- [70] Decision Tree Classifier. *Dinhanhthi.com* [online]. [cit. 2023-03-27]. Available from: <https://dinhhanhthi.com/decision-tree-classifier/>
- [71] LUKAS F., Meier, 2022. *ANOVA and Mixed Models*. Taylor & Francis. ISBN 9780367704209.
- [72] PyTorch. *Pytorch.org* [online]. [cit. 2023-03-27]. Available from: <https://pytorch.org>
- [73] TensorFlow. *Www.tensorflow.org* [online]. [cit. 2023-03-27]. Available from: https://www.tensorflow.org/?gclid=CjwKCAjw_YShBhAiEiwAMomsELVs292Bv-c7LhlOqjRiSsufhzOkfk1cRS4ZSMEEtLF1zwvvIN7C-BoC6sgQAvD_BwE

- [74] Scikit-learn. *Scikit-learn.org* [online]. [cit. 2023-03-27]. Available from: <https://scikit-learn.org/stable/>
- [75] Cyclical Learning Rates — The ultimate guide for setting learning rates for Neural Networks. *Medium.com* [online]. [cit. 2023-03-27]. Available from: <https://medium.com/swlh/cyclical-learning-rates-the-ultimate-guide-for-setting-learning-rates-for-neural-networks-3104e906f0ae>
- [76] *Training data-efficient image transformers & distillation through attention* [online]. [cit. 2023-03-27]. Available from: doi:10.48550/arXiv.2012.12877
- [77] HE, Kaiming, Xiangyu ZHANG, Shaoqing REN a Jian SUN. Deep Residual Learning for Image Recognition. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* [online]. IEEE, 2016, 2016, s. 770-778 [cit. 2023-03-27]. ISBN 978-1-4673-8851-1. Available from: doi:10.1109/CVPR.2016.90
- [78] HUANG, Gao, Zhuang LIU, Laurens VAN DER MAATEN a Kilian Q. WEINBERGER. Densely Connected Convolutional Networks. In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* [online]. IEEE, 2017, 2017, s. 2261-2269 [cit. 2023-03-27]. ISBN 978-1-5386-0457-1. Available from: doi:10.1109/CVPR.2017.243
- [79] HINTON, Geoffrey, Oriol VINYALS a Jeff DEAN, 2015. *Distilling the Knowledge in a Neural Network* [online]. [cit. 2023-03-27]. Available from: doi:10.48550/arXiv.1503.02531

LIST OF ABBREVIATIONS

APK Android Application Package

CNN Convolution neural network

VDR vector detection ratio

DVM Dalvik virtual machine

HAL Hardware Abstract Layer

ART Android runtime

NDK Native Development Kit

JNI Java Native Environment

MLP multi-layer perceptron

ANN artificial neural network

NLP natural language processing

LSTM Long Short-Term memory

ViT Vision Transformer

ML machine learning

MB Megabytes

SVM Support Vector Machine

DTC Decision Tree Classifier

Gbrt gradient boosted regression tree method

DeiT Data-efficient image transformers

GELU Gaussian Error Linear Units

ReLu Rectified Linear Unit

MD-NNM Malware detection neural network model

MD-ViTNNM Malware detection vision transformer neural network model

GPU Graphics processing unit

TPE tree Parzen estimator

GUI graphical user interface

LIST OF FIGURES

Figure 1. <i>Android Layer cake</i> [36].	19
Figure 2. <i>Dalvik Virtual Machine</i> [42].	20
Figure 3. <i>Android Run Time</i> [38].	21
Figure 4. <i>Android components</i> .	23
Figure 5. <i>Android APK structure</i> .	25
Figure 6. <i>Illustration of a biologic and artificial neural network</i> .	28
Figure 7. <i>An artificial neuron</i> .	30
Figure 8. <i>Neural network architecture</i> .	30
Figure 9. <i>Gradient descent</i> .	32
Figure 10. <i>Activation functions</i> [50].	33
Figure 11. <i>Convolution operation</i> .	34
Figure 12. <i>Max pooling operation</i> .	35
Figure 13. <i>Architecture of CNN</i> .	36
Figure 14. <i>Transformer architecture</i> [23].	38
Figure 15. <i>Scaled Dot-Product Attention and Multi-Head Attention</i> [26].	39
Figure 16. <i>Vision Transformer (ViT)</i> [27].	40
Figure 17. <i>Size analysis of the clean and malicious applications</i> .	47
Figure 18. <i>Splitting of the dataset</i> .	48
Figure 19. <i>Simplified methodology of APK to greyscale image</i> .	51
Figure 20. <i>Samples of grayscale images for "APK to grayscale image" approach</i> .	52
Figure 21. <i>simplified methodology of DEX to greyscale image</i> .	53
Figure 22. <i>Samples of grayscale images for "DEX to grayscale image" approach</i> .	54
Figure 23. <i>Simplified methodology of DEX to RGB image</i> .	55
Figure 24. <i>Samples of RGB images for "DEX to RGB image" approach</i> .	57
Figure 25. <i>simplified methodology of extracting properties of AndroidManifest.xml</i> .	58
Figure 26. <i>Feature selection process</i> [69, 70].	60
Figure 27. <i>Display of optimization results for testing approaches</i> .	62
Figure 28. <i>Architecture of MD-NNM</i> .	65
Figure 29. <i>The showcase of learning process for MD-NNM</i> .	69
Figure 30. <i>Architecture of MD-ViTNNM</i> .	71
Figure 31. <i>showcase of learning process for MD-ViTNNM</i> .	73
Figure 32. <i>Optimization process</i> .	76
Figure 33. <i>Results of the optimization process</i> .	76
Figure 34. <i>Project architecture in the PyCharm environment</i> .	78

Figure 35 <i>Graphical interface of the project</i>	79
Figure 36 Possible Graphical Interface Settings	80
Figure 37 Testing result.	81

LIST OF TABLES

Table 1. <i>Set of defined questions.</i>	12
Table 2. <i>Showcase of tools [11].</i>	14
Table 3. <i>Selection of relevant scientific papers.</i>	16
Table 4. <i>Summary of app distribution [35].</i>	17
Table 5. <i>Illustration of possible hyper-parameters.</i>	42
Table 6. <i>Dataset overview</i>	46
Table 7. <i>Subset summary.</i>	49
Table 8. <i>A summary of feature extraction technologies for Android applications.</i>	50
Table 9. <i>Pseudo code that represents the conversion of an Android application to a grayscale image.</i>	51
Table 10. <i>Pseudo code that represents the conversion of DEX to a grayscale image.</i>	53
Table 11. <i>Pseudo code that represents the conversion of DEX to a RGB image.</i>	56
Table 12. <i>Pseudo code that represents the extraction of AndroidManifest.xml properties.</i>	58
Table 13. <i>Summary of Android application preprocessing approaches.</i>	59
Table 14. <i>Hyper-parameter optimization grid for feature testing.</i>	61
Table 15. <i>Results of Tukey test.</i>	63
Table 16. <i>Randomly selected MD-NNM parameters.</i>	68
Table 17. <i>Accuracy and Confusion matrix of test process of MD-NNM.</i>	69
Table 18. <i>Randomly selected MD-ViTNNM parameters.</i>	72
Table 19. <i>Accuracy and Confusion matrix of test process of MD-ViTNNM.</i>	73
Table 20 <i>Hyper optimization parameters</i>	74

LIST OF APPENDICES

1, malware_detection_system.zip