TDK thesis

Malware classification using neural network-based image processing methods

Levente Imre Dobák

Advisors:
Dr. Gábor Hullám, Mihály Vetró

October 31. 2022

Department of Measurement and Information Systems
Budapest University of Technology and Economics

Contents

1 Abstract

Nowadays, there is an increasing focus on cybersecurity, including the effective prediction and filtering of the evolution of malicious software, or malware in other words. The size of the problem is illustrated by the AV-TEST Institute's estimate that between 4,5 and 5,6 hundred thousand new malicious programs and software are created every day. This poses a huge threat to public organizations, economic operators (e.g., financial institutions, enterprises) and even ordinary users. Malware that infiltrates our systems and installs itself on our devices can even put key critical data in the wrong hands.

Unfortunately, in the constant battle to protect ourselves from threats that affect our security, the evolution of malware in cyberspace today has significantly outpaced our ability to defend against them. Several methods are being tried to improve protection, but one of the most effective has been proven to be artificial intelligence-based behavioural analysis. Two types of analysis are commonly used, dynamic and static, but the specific techniques differ considerably. Dynamic analysis has a higher chance of filtering out so-called day 0 (new) malware, but it is more difficult to perform, so the more common static methods are mostly used to predict and filter out patterns.

In this paper, I present a novel model architecture based on static analysis using the Microsoft Malware Classification Challenge (BIG 2015) database. The model consists of two parts, which have been tested separately. The first and most widespread approach processes malware bytecodes as grayscale images, while the second one is based purely on bytecode analysis.

In this study, I try to use the combined power of these two models to create a more efficient method with a higher success rate. I convert malware bytecodes into grayscale-images, which I use to build classification models. The uniqueness of the method lies in the fact that I use two different neural networks to first process the image-converted dataset and then do the same with the bytecode in the second case. The results obtained are compared with those found in the literature.

2 Introduction

Nowadays, the use of some kind of computing device is almost indispensable in everyday life. As technology evolves, so do these devices, and more and more of them are flooding the market at affordable prices. The big advantage of this is that it becomes accessible to many and makes our daily lives easier. However, most users do not have a deep knowledge of information technology, which makes it easier for them to become victims of threats that can pose serious risks to the everyday user.

Over the past decades, not only devices and the operating systems and software that run on them have evolved, but also the malware that can be installed on them. It is part of our lives that there is a constant battle between hackers/cybercriminals and IT professionals to detect and prevent the entry of these malware in time. Many technologies have been developed to do this, such as heuristics-based malware detection, which is based on observing normal and abnormal system behaviour.

In my thesis, however, I will focus on a different kind of technological approach, which will be signature-based malware detection, which is based on the bit sequences, also known as signatures, of malware in an attempt to detect and classify the malware and its family.

There are three main techniques to analyse malware:

- dynamic analysis
- hybrid analysis
- static analysis

In dynamic analysis[20], the malicious program is analysed while being executed. This means looking at several factors while the program is running, including for instance function calls or even the parameters given to the function. Although this method is more accurate than static analysis, it is considerably more difficult to perform. It requires the use of various tools, such as a sandbox environment, or RegShot[20] emulators. In addition, sophisticated malwares can even detect when such programs are running in the background, with the potential risk of the malware being disguised as a benign program, making detection more challenging.

Static analysis, also known as code analysis, unlike dynamic analysis, does not require the program to be executed. During the analysis, the byte codes of the program are used to search for possible patterns of unwanted behaviour. Various tools can be used to implement the technique, such as a debugger or a source code analyser[20].

Hybrid analysis, as the name implies, combines the power of both techniques to effectively filter malware. The advantage of this is that we can use the positive aspects of both techniques defined above. The analysis begins with static analysis and then proceeds to dynamic analysis once the right patterns are found, where we follow the same procedure as described above.[20]

In this study, I will apply the technique of static analysis, the key point of which, as I mentioned above, is that the software or code fragment is not executed. This allows us to analyse malicious software more safely and quickly.

My ambition was to develop a unique, previously unexplored method based on the combination of two well established approaches (image-based static analysis and byte-code-based static analysis). In the next chapter, I will explain the basic definitions and basic methods that have helped me to prepare my research.

This thesis is structured as follows. In chapter three and four the theoretical background and the basic definitions of malwares and deep learning is explained.

The sixth chapter contains the description of the dataset I have used. I introduce the applied solutions in the following chapters, where I describe in detail the method and solution of image-based analysis, and finally, I demonstrate my solution implemented using byte-code-based analysis. A full description of the implemented model is presented afterwards.

In the eighth chapter, the results achieved are analysed and compared, and a conclusion is presented about the effectiveness of my method. In addition, some options for further development are discussed which were beyond the scope of this TDK paper, but could be worth examining.

3 What is Malware and what types are known

3.1 Definition of malware

One of the key focuses of my thesis is malware, but what is malware really?

Malware is a „Software or firmware intended to perform an unauthorized process that will have adverse impact on the confidentiality, integrity, or availability of an information system."[12]

There are many known types, so it would be beyond the scope of this paper to list them all, but I do believe it is important to describe some of the main types, so I will provide more detail on the types I consider important without claiming completeness. To formulate the description of the main malware types, I have used the resources made by MalwareBytes[11].

3.2 Trojan

The name is quite revealing, surely everyone knows the story of the city of Troy, where the Trojan horse entered the city. It's a kind of strategic trick that allows the malware to easily get through security gaps. The malware is also named after this strategy/trick, because its behaviour is quite similar to this trick.

The Trojan malware infiltrates the victim's computer by deceiving people and using various social engineering techniques. Once inside, it can easily duplicate itself, inserting its content into other programs and flooding the targeted computer.

The virus first appeared in 1975, the year in which a program (ANIMAL) was released, which led to a large number of people falling victim to this virus. It kept evolving, so new methods of injecting malware were developed.

3.3 Trojan Downloader

A trojan downloader, or just plain downloader as the name suggests, is a malware whose main purpose is to help further malware to enter the system. It will do this by installing other, unrelated software on the victim's already infected computer.

3.4 Worm

The worm is a rather tricky, invasive type of malware that is not always under human control. The worm is a kind of trojan malware, once it penetrates, it can easily duplicate/multiply itself, achieving full infection.

The main threat is that replication happens on its own, without any external intervention (our execution), so we can fall victim to it even through no fault of our own. It also has the down side that it can easily spread from one computer to another via the network. There are several known methods to gain first access, to mention a few:

- Phising
- Networks
- File sharing
- External devices

3.5 Adware

This type of malware is less harmful than those previously mentioned. Adware can be more accurately described as an undesirable, annoying program, but in itself it does not do any damage to the infected device.

The purpose of the adware is to flood the owner of the infected device with advertisements once it has been infiltrated. At first, not necessarily with relevant ads, but later on, adware can analyse the user's browsing habits and browsing history, so they can even display ads in a targeted way. This is beneficial to them because it generates illegal revenue for adware developers.

Unfortunately, it is easy to become a victim, because adware is commonly installed alongside free software, or "freeware"[11] that is considered to be safe. Adware is browser independent, so it can be found in any browser (edge, chrome, mozilla, opera, safari). This type of malware has been detected since about 1995 and is constantly evolving.

3.6 Backdoor

A backdoor is actually a vulnerability on our devices through which certain malware can gain access. We can think of it as an unprotected backdoor through which one can easily "rob the bank" without being noticed. Unfortunately, some of these loopholes are hard to prevent as they are used not only for malicious purposes. In some cases they can be used to recover an account from which we have completely locked ourselves out.

The purpose of the malware that enters our device through this so-called "backdoor" is to gain full access to our data without us as a user realizing

it. Thus, thanks to full access, they can easily and repeatedly steal our data without us noticing, or use our data as a basis for future blackmail.

3.7 Spyware

As the name may imply, these malwares, which are of the spyware type, operate stealthily, but effectively in the background without our knowledge. They are designed to steal our personal data, monitor our behaviour, and possibly steal crucial passwords and code.

There are many different ways to break in, one of which is through the aforementioned backdoor, which allows them to remain completely invisible. But they can also sneak in by disguising themselves as an efficient program that the user can install on their computer, or even by using phishing emails, where a careless click can infect the targeted computer.

3.8 Ransomware

Ransom malware, or simply ransomware, is perhaps the most impactful and most noticeable malware of the types listed here. There are several possible ways to deliver the malware into the target computer, such as:

- „Malspam”
- „Malvertising”
- „Spear phishing”
- „Social engineering”

There are several subtypes of ransomware. Its aim is to extort money from its victim after gaining access, which it achieves in various ways. Some of the subtypes are:

- „Scareware”
- „Screen lockers”
- „Encrypting ransomware”

The mechanics of scareware is to "scare" the user by making statements that are not true. For instance, a virus may have been found on the computer, but the user must pay to have it removed.

Screen lockers, as the name implies, lock the user out of their account and won't let them in even after repeated attempts until they comply and pay.

Encrypting ransomware's technique is to steal and then encrypt our files, so we are definitely not granted access until we pay the demand. However, it's not necessarily a good idea to settle the claim, as there is no guarantee that we will be able to access our files afterwards.

3.9 Obfuscator

It is important to mention obfuscator, even though obfuscator is not a specific type of malware, but rather techniques that allow polymorphic malware, i.e. malware that can regenerate the decryptor needed to recover its encrypted code each time, to disguise itself more easily, thus reducing the risk of being detected.[22]
There are several obfuscation techniques, the best known of which are:

- Dead-Code Insertion

The dead-code insertion technique works in a way that is perhaps implied by its name, by adding to the existing useful code with a given behaviour, a so-called dead code that has no function. Once this code is inserted, the original code does not change its behaviour, only the structure of the malware changes, making it more difficult to identify.

- Register Reassignment

The main idea of this technique is that the program code stored in the registers is packed into a different register each time it is re-executed, without changing the behaviour.

- Subroutine Reordering

Subroutine reordering, as the name indicates, swaps the order of subroutine calls at runtime, without changing the behaviour.

- Instruction Substitution

It's a very simple technique that involves replacing some commands with equivalent commands, so it will not change the behaviour, but it will modify the structure.

- Code Transposition

Rearranges the sequences of the code.

- Code Integration

Last but not least, a rather interesting technique is that the malware attaches itself to the chosen target program, making detection much more difficult.

4 Deep Learning

As I have mentioned previously, there are many different approaches to filtering malware. After a thorough review of the literature, there are many authors who discuss different, individual methods. The usefulness and effectiveness of each method varies over a wide spectrum.

In the following deep learning and classification methods are explained in a bit more detailed way, as they are among the most commonly used methods.

4.1 The history of deep learning

The very first deep learning algorithm, which was already somewhat similar to the one we use today, dates back to 1965. Ivakhnenko and Lapa were the first to create this model.[2]

Although very different from the current models, this early model also had several layers, but was much thinner than its modern counterparts. They used the polynomial activation function in their model.

However, the first convolutional networks, which were already very similar to those being used today, were not used until 1979, when Fukushima created a model with several convolutional and pooling layers.

Although machine learning was very popular during these decades and a considerable amount of money was spent on its research and development, it did not actually improve at the same pace as nowadays due to a lack of adequate computational hardware.

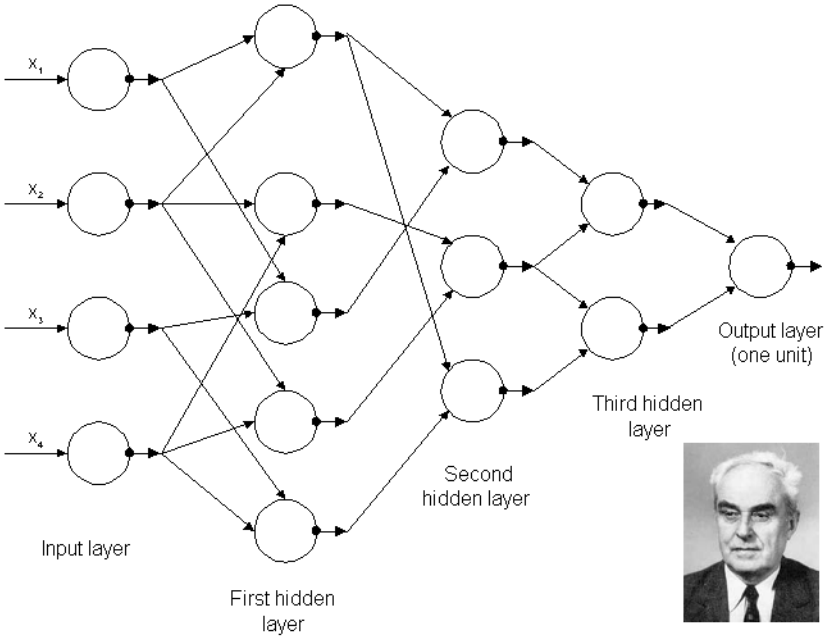

Figure 1: Ivakhnenko's model[2][14]

4.2 Deep Learning and Neural Networks

Deep learning is a branch of artificial intelligence that allows us to model complex phenomena using advanced architectures, such as deep neural networks, and efficient learning methods.
The resulting model performs various non-linear transformations on the data using a complex web of neurons which form a neural network. The neuron is also known as a processing unit. They are grouped into multiple layers, such that each layer contains multiple neurons.
Their application can have many specializations, such as speech recognition or, as the subject of my thesis, image recognition. Neural networks are already well known from biology, and this knowledge was the inspiration for the construction of these models.
In the following, the above-mentioned layers will be discussed, the concepts of input layer, hidden layer and output layer will be introduced, and optimization, batch and epoch will be explained to provide a basic overview.[5][7]

4.3 Input layer

The operational principle and role of the input layer is very simple. It is the layer that receives the input datapoints to be processed and transfers it to the next layer.

This layer is also made up of neurons, the quantity of which is determined by the dimensionality of the input data. As an illustration, suppose that the

input matrix has the shape (64000, 9), in which case the number of neurons in our input layer will be nine. These neurons are connected one by one with different weight values to each neuron of the next layer.

4.4 Hidden layer

The size of the hidden layer is not necessarily the same as the size of the input layer, since each neuron of the hidden layer is connected to each neuron of the input layer.

Several hidden layers can be utilized in the design of a model, there is no specific golden rule for the appropriate number. It depends on many factors how many layers will provide adequate the results.

The more layers used, the "deeper" the neural network is considered. Deeper models are capable of approximating more complex functions over the input data, however an overly complex model, i.e. a model with too many layers, is prone to overfitting, depending on the quantity of the available data and the difficulty of the function that is to be approximated.

4.5 Output layer

The output layer, as the name implies, produces the calculated outputs. These outputs are optimized during training, and activation functions are usually applied to the output layer to constrain its value into the known value set of the data.

4.6 Loss

During the learning process, we want to achieve the best possible results, but we can assume that there will always be some amount of error in the output. This error comes from the difference between the predicted output and the actual output. This difference also serves as the basis of the target function of the optimization process, we call “learning”. This target function is most commonly called the loss function, and the main goal of the learning process is to find the global minima of this function over the parameters of the model.

4.7 Batch and epoch

These definitions are often used in the training, so I thought it was worth mentioning.

A batch is a subset of the dataset that is used to perform a single adjustment on the model weights, based on the aggregated gradient of each individual data point inside the batch. During teaching, the dataset is not taken as input in

one piece, but is loaded in smaller 'batches'.

An epoch is an indicator of a complete iteration, when the entire content of our dataset (or equivalent amount of data points) has been used for teaching using the aforementioned batches.

4.8 Activation functions

The use of activation functions is essential in a complex model, as they allow our model to approximate a wide range of functions (e.g. complex logical functions), which would not be possible with only linear activations.

Although solving a linear equation can be facilitated more efficiently, a linear model will not be able to represent more complex patterns. In the following, I will describe the most relevant and commonly used activation functions that I have also used to generate my models.[3][7][15][17][18]

4.8.1 Sigmoid

The sigmoid activation function is probably the most frequently used function. It owes its success to the fact that its derivation is easy, and it can be continuously derived.

Its derivative:

$$f'(x) = 1 - sigmoid(x) \tag{1}$$

The sigmoid is not symmetric with respect to the zero point, and its output will be a number between 0 and 1:

$$f(x) = \frac{1}{e^{-x}} \tag{2}$$

4.8.2 ReLU

A characteristic of the Rectified Linear Unit, or ReLU, which will appear frequently in my thesis, is that it is a non-linear activation function that enables the model to "turn off" the output of certain neurons for a given input, essentially implementing a logical function of arbitrary complexity. It can be defined in the following way:

$$f(x) = max(0, x) \tag{3}$$

4.8.3 LeakyReLU

It is very similar to the ReLU function, with the only difference being that LeakyReLU allows a negative value for x. Therefore, if x is negative, it replaces this value with a very small multiplication of x.

The definition of the LeakyReLU function is shown below:

$$f(x) = x, x >= 0 \tag{4}$$

$$f(x) = 0.01x, x < 0 \tag{5}$$

4.8.4 Parametrized ReLU

An alternative approach to the LeakyReLU function where x is not replaced by 0.01x for negative x, but by an introduced Epsilon value. Its efficiency is a consequence of this, and is defined as follows:

$$f(x) = x, x >= 0 \tag{6}$$

$$f(x) = \varepsilon x, x < 0 \tag{7}$$

4.8.5 Swish

A relatively new activation function introduced by GOOGLE, where the main feature is that the input and the output can vary in opposite directions. Even though the input increases the output of the function can still be lower.

$$f(x) = x * sigmoid(x) \tag{8}$$

$$f(x) = \frac{x}{1 - e^{-x}} \tag{9}$$

4.8.6 SoftMax

The main feature of the SoftMax function is that it can be used for classification for multiple categories. It is a combination of several sigmoid functions[17], where its return value will take the form of a probability, which represents the probability that a given input belongs to a particular class.

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad for\ j = 1, ..., K. \tag{10}$$

4.9 Deep Feedforward Networks

The deep feedforward network is also often referred to as a feedforward neural network, because it is also based on the principles of neuroscience discussed in the previous section. In general, the operation of feedforward networks is based on the principle that we want to estimate a given function. This can be done, for example, in the following way: let y be a category for which the given input x is mapped by the function.[5]

This mapping is done according to the following function, illustrated below.

The θ in the function is an arbitrary parameter, and we are responsible for determining its value. The options to specify its exact value will be discussed in more detail later.

$$y = f(x; \theta) \tag{11}$$

The feedforward network is named after the fact that the input x value is processed by a predefined function f, whose output is y, and this value is then fed into subsequent calculations.

A feedforward network can be composed of multiple layers as already explained above. As I have mentioned previously, we have to determine the value of the parameter θ and there are several possible options to define it:

- Our first and perhaps simplest option is to take the value of a completely general infinite-dimensional Φ, but this is not the best solution, as it is too general and therefore not the most efficient for solving more complex problems.
- Alternatively, we can calculate the value of Φ ourselves. Unfortunately, this is very time-consuming so not very practical. Thus, this is not the most sufficient option to choose.
- A possible third option is to apply a strategy whereby we do not determine the value, but rather 'learn' the value. To implement this, we are using a model such as:

$$y = f(x; \theta, \omega) = \Phi(x; \theta)^T \omega. \tag{12}$$

We now have a parameter (θ) that can be used to determine Φ. With the help of the parameter ω we can map this from Φ(x) to the output we are looking for.

The feedforward network is a key element of machine learning, as it is also the core of the convolutional network, which is a special format of the feedforward network. I have devoted a separate subsection to the convolutional network, which is discussed in the next section below.

4.10 Convolutional networks

CNN, or convolutional neural network, owes its popularity to its ability to search and process patterns in image data efficiently and easily.

Its structure is usually made up of four different layers:[5][6]

4.10.1 Input layer

The input layer performs its role as described above, however, in the case of a convolutional neural network, the input data is not necessarily converted into

a one-dimensional vector but can also be processed in two-dimensions. Thus, we can effortlessly give it, for example, a 64x64 pixel image in the form of a 64x64 matrix, which I will do in one of my models later. The advantage of this is that the relationships between each element can be more accurately expressed using the matrix.

4.10.2 Convolution layer

We can think of the convolution layer as a kind of filter, and we can state that, for a two-dimensional image, it will remain a two-dimensional image after processing. Its operation can be summed up as follows: get a 64x64 image, each pixel of which can be described by a number between 0 and 255. This picture is passed to the layer. The layer can be thought of as a small image-like filter similar to the image. Assume that its size in this example is 3x3. In our resulting filter, we set all nine values to 0 or 1 in a completely random manner. Next, we apply the filter to our image, in a way that the top left corner of our filter is the same as the top left corner of the image we wish to study, thus covering an area as large as our filter. Once this has been done, the number in the filter, which is 0 or 1, is multiplied by the pixel value of the image (in this case, the number 0-255), this is done for each pixel covered by our filter. In the next step, the values are summed, forming a single pixel instead of the nine we had before. Once this has been done, we move our filter one pixel to the right and follow the same technique until we reach the edge of the image, at which point we move it back to the left of the image one pixel down. These steps are continued until the entire image is mapped.

To determine the behaviour and structure of the filter, we have various properties, such as:[6]

- Filter size

The size of the filter in our example was 3x3, but of course the size is adjustable, so it can be 5x5 as well. Choosing a size too large can significantly increase computational complexity.

- Padding

If we want to keep the size of our output image unchanged, we can also use padding, which allows us to enlarge our original input image by adding as many pixels with a value of 0 around its circumference as we specify for the padding value.

- Stride

In the technique described above, we always shifted our filter by a single pixel to the right or down, but this is not mandatory either. With stride we can specify the amount of lateral and vertical offset of our filter.

- Dilation

By specifying dilation, we can increase a 3x3 filter to 5x5, for example, by always skipping a pixel in our analysis. Thus, although the number of pixels analysed remains the same (e.g. nine), this operation is done on a 5x5 area of twenty-five pixels.

- Activation function

Last but not least, the results obtained can be further transformed using an activation function. Any of the activation functions described above can be used. The most commonly used is the ReLU function.

4.10.3 Pooling layer

The pooling layer is usually applied after the convolution layer, and its behaviour is very similar to it. It is also a kind of filter and maps the data in the same way as the convolution layer. The difference between the two is that the pooling layer picks the element with the highest value (in case of MaxPooling) from the area covered and keeps only that. This reduces the dimensionality of the image, thus reducing the computational requirements, which speeds up the learning process.

4.10.4 Fully connected layer

The fully connected layer is the final output layer that connects the nodes from the previous layers. The resulting number of outputs is equal to the number of classes expected in the classification.

5 Different approaches in the literature

As I mentioned, there are many approaches to classifying malware. Looking at the literature, there are many authors who have discussed different methods. The effectiveness of each method varies over a wide spectrum. In the following, I will describe in detail some of these methods from some authors that provided inspiration for this paper.

5.1 Hybrid framework

An interesting approach to the detection of malware is shown in Stephen O'Shaughnessy's and Stephen Sheridan's article[13], which has the peculiarity of approaching the analysis of malicious software from two directions. First, it relies on the static analysis method on which I have based my work, but this is complemented by a very interesting dynamic analysis. It processes executable malware using a virtual machine (VMI). First, it executes the relevant program code on the VMI, and then subsequently writes the process memory dump extracted during the execution back to the host computer's disk. In the second step, the two different data are combined and converted to SFC(System File Checker) format using the scurve library. In the third step, three feature description algorithms are evaluated.

These are the following algorithms:

- Local Binary Patterns (LBP)
- Gabor Filter
- Histogram of Oriented Gradients (HOG)

The already prepared SFC format images are also tested with three different classification algorithms.

- Random Forest (RF)
- Support Vector Machine (SVM)
- K-nearest Neighbours (KNN)

The study highlights that higher image dimension numbers can be used to obtain more accurate results, after appropriate power tuning. With an image dimension number of 512, a very high performance of 97.9% accuracy in classification was achieved.

5.2 Malware Classification by Binary Sequences

In their paper[10], Wei-Cheng Lin and Yi-Ren Yeh discuss classification based on byte-code sequences as one of the building blocks of my Mixed-Model, where they demonstrate that after proper optimization, a one-dimensional byte array can be processed more efficiently and with similar precision as the formatted image data.

The comparison is based on an algorithm for classifying two-dimensional images of various sizes generated from byte code, as a competitor to the method they developed and explained.

The paper discusses in detail the steps of the transformation to be applied to the byte code. Such important details include, for example, how the byte codes can be assigned one by one to the resolution of the two-dimensional images generated from them.

The study uses the Microsoft Malware Classification Challenge (BIG 2015) dataset.

Their approach involved six convolutional blocks, connected by two fully connected layers. Using this model, after appropriate optimizations, a result of 96.32-98.7% was obtained on the 1x2034 byte sequence.

Although this is considered somewhat lower than two-dimensional image-based classification, it may be a more practical choice in many cases. This is because it can greatly reduce the time and potential cost of the learning procedure.

5.3 Image-based classification methods

The vast majority of studies in the literature are based on two-dimensional image-based static analysis, which is one of the most widely investigated methods today for effective filtering and classification of malware.

There are many different approaches[1][8][19][20][23] in this field of research, the main difference being in finding the right image format. Finding the optimal resolution of images is a very difficult challenge for researchers. In many cases, this depends on the efficient transformation of images, for which there are also many methods available. As a result, the difference in the results obtained may even come from this.

Since the byte codes available for each malware have different sizes, thus the images produced from them will have differing resolutions and shapes as well. The resulting images can only be used effectively after some standardization procedure. The aim is to achieve a uniform size over all samples.

Most studies use images with the same width and height, achieving over 90% accuracy in classification. But there are also studies where these parameters differ.

One example is Tran The Son's, Chando Lee's, Hoa Le-Minh's, Nauman Aslam's and Vuong Cong Dat's article, where both the latitude factors and the height factors are powers of two, but their values are not equal.

In the present case, they used 8x64, 16x64, 32x64 resolutions for their investigations. One of the main reasons for this, which I have observed in my work, is that the images generated from the byte code are nowhere near square. This makes it easy to rescale their width parameter. However, the images need to be converted to a uniform fixed size, otherwise it is difficult to recognize the patterns correctly during learning. Two types of machine learning methods were used in this study:

- k-NN (k nearest Neighbour)
- CNN (Convolutional neural network)

Their results were prominent, as their study demonstrated that a similar 97.8% accuracy can be achieved using images with different parameters than with normalized 64x64 images. They also experienced that the smaller the image size, the more loss was observed during classification.

6 Used resources and software

6.1 Software

I prepared the source code for this study in the Python programming language, which is also available on github at the link marked in the reference list [4]. I used the PyCharm IDE, provided by JetBrains, to transform the data properly, and the Jupyter Notebook framework to build and train the model.

6.2 Resources

The hardware specifications of the computer on which the models were executed and the tests were performed consisted of the following. An eight core intel core i7-9700K CPU with a base clock speed of 3.60GHz. As for the GPU, an NVIDIA GeForce GTX 1660 model was used, and finally 16GB of system memory.

6.3 Description of the dataset

6.3.1 Dataset

Finding the appropriate dataset was the most challenging starting point for my TDK thesis. I chose the Microsoft Malware Classification Challenge (BIG 2015) dataset published by Microsoft in 2015. The dataset is freely available at www.kaggle.com.

6.3.2 File extensions

The dataset contains files with different extensions but with semantically identical content. The files with the ".asm", contain metadata such as function calls, and they were generated by the IDA disassembler tool. The more relevant extension for this thesis is ".bytes", which contains the binary data without the headers. This is necessary to make the malware sterile, i.e. harmless.

6.3.3 The structure of the dataset

The downloaded and unpacked data (about half a terabyte) can be split into three parts. It contains an excel spreadsheet with an .xls extension and two subfolders, defined as a train folder and a test folder.

6.3.4 Train Label

The content of the trainLabels file is composed of key-value pairs, each of which contains the corresponding classes (value) for each file based on the file names (key) that we use during our training. Opening our the file, we can see the following:

Figure 2: An example of the content of the train label

6.3.5 Train

The train folder contains 10868 .bytes and the same number of .asm files that can be matched to the bytes files. Each of these files can be considered a piece

of malware. The malwares in the folder can be classified into nine different groups.[16]

Their distribution is as follows:

- There are 1,541 instances of malware of the Ramnit family in the database. Their type is the Worm described in the chapter above
- There are 2,478 instances of malware of the Lollipop family in the database. Their type is the Adware described in the chapter above
- There are 2,942 instances of malware of the Kelihos_ver3 family in the database. Their type is the Backdoor described in the chapter above
- There are 475 instances of malware of the Vundo family in the database. Their type is the Trojan described in the chapter above
- There are 42 instances of the Simda family of malware in the database, of the Backdoor type described above, making Simda the most under-represented malware in the database.
- There are 751 instances of malware of the Tracur family in the database. Their type is the TrojanDownloader described in the chapter above
- There are 398 instances of malware of the Kelihos_ver1 family in the database. Their type is the Backdoor described in the chapter above
- There are 1,228 instances of malware of the Obfuscator.ACY family in the database. Their type is the obfuscator described in the chapter above
- And finally, of the Gatak family of malware, there are 1013 instances in the database, of the type Backdoor described above

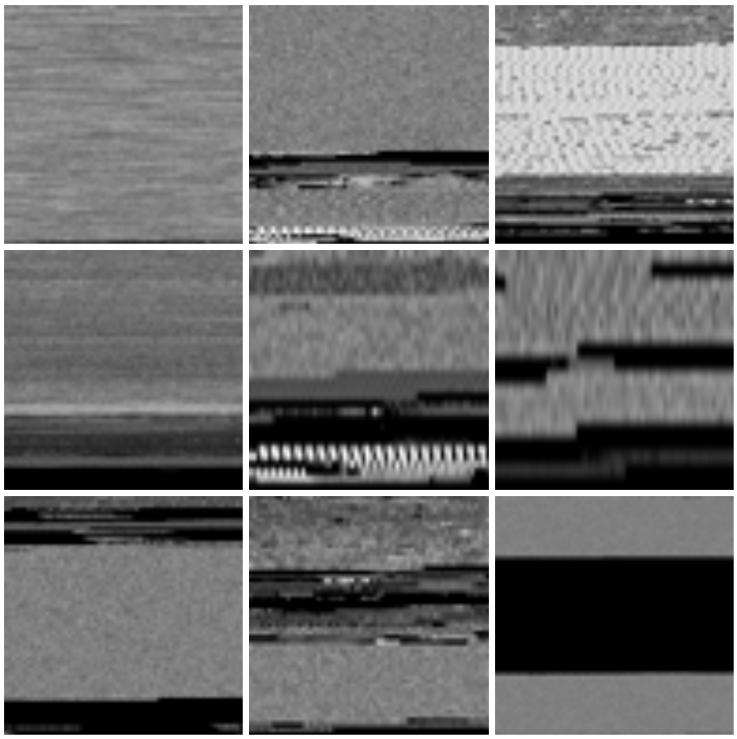

Figure 3: An example of all malware families

6.3.6 Test

The test folder contains malwares without any kind of label, which can be used to test our model after being trained. The test directory contains 10873 pieces of malware.

7 About the models used

7.1 The image-based classification model

7.1.1 Specific features and brief description of the model

From the information above, it is already clear that the image-based approach to static analysis is the most widely investigated methodology today. Therefore, I think it is important to start the explanation of the classification models I have constructed with a description of this method.

The effectiveness of the method can be attributed to a number of reasons, but perhaps the most prevalent is that image processing algorithms can recognize global patterns in the source code, and they can also process information from the entirety of the bytes file, after the input is standardized through rescaling.

The model I have composed processes the above-mentioned dataset and classifies it according to the nine classes found in the database. However, to achieve this I first needed to completely transform the structure and extension of the files in the dataset.

The transformation steps and their motivation are described in detail in the next paragraph.

7.1.2 Processing the raw dataset

Since our dataset does not store data in an image format, I had to convert it and generate the images required for efficient processing. Since there is no well-established way on how to properly transform bytecode into an image, and I only found recommendations on the resolution of the image, I had to figure out and test the transformation technique myself. I will describe the steps of this in the following.

I worked with the content of the bytes files, as it represented the relevant information for me. An example to the original content of our bytes file is shown in figure 4.

```
00401000 E8 0B 00 00 00 E9 16 00 00 00 90 90 90 90 90 90
00401010 B9 25 2B 56 00 FF 25 80 23 41 00 90 90 90 90 90
00401020 68 30 10 40 00 E8 70 03 01 00 59 C3 90 90 90 90
00401030 B9 25 2B 56 00 FF 25 74 23 41 00 90 90 90 90 90
00401040 E8 0B 00 00 00 E9 16 00 00 00 90 90 90 90 90 90
00401050 B9 24 2B 56 00 FF 25 78 23 41 00 90 90 90 90 90
00401060 68 70 10 40 00 E8 30 03 01 00 59 C3 90 90 90 90
00401070 B9 24 2B 56 00 FF 25 7C 23 41 00 90 90 90 90 90
```

Figure 4: An example of the original file content

This explains to us that the structure of our file consists of:

- A line identifier at the beginning of each line, which has no meaningful role for the malware.
- The rest of the line indicates sixteen bytes stored in hexadecimal format.

When cleaning the contents of the raw file, I first deleted the row identifiers from the beginning of each row. This has no influence on the structure of the malware, so it can be safely deleted. After that, only the byte codes remained.

```
E8 0B 00 00 00 E9 16 00 00 00 90 90 90 90 90 90
B9 25 2B 56 00 FF 25 80 23 41 00 90 90 90 90 90
68 30 10 40 00 E8 70 03 01 00 59 C3 90 90 90 90
B9 25 2B 56 00 FF 25 74 23 41 00 90 90 90 90 90
E8 0B 00 00 00 E9 16 00 00 00 90 90 90 90 90 90
```

Figure 5: An example of the file content without the row identifiers

To create a grayscale image, I have equated the value of a byte to the value of a pixel. This value should be a number between 0 and 255. Although this is satisfied in hexadecimal form in this file, the PIL library I used can only form images using decimal numbers. Consequently, hexadecimal numbers were converted to decimal numbers in the next step, as it can be seen in figure 6.

```
255 133 248 248 248 248 248 255 21 96 48 84 248 248 74 248
248 116 255 248 248 248 139 248 248 248 248 248 248 116 232 248
255 248 86 248 248 248 89 248 248 248 248 255 139 232 248 248
255 255 80 248 248 248 248 255 248 248 248 73 248 255 21 28
49 84 248 248 248 248 248 6 255 248 248 248 248 248 133 89
248 248 248 248 248 255 21 136 49 84 255 255 139 133 255 117
```

Figure 6: An example of the file content after the modifications

I have put the contents of the resulting file into a numpy array, which I reshaped into a one-dimensional vector.

Then, I have set the image width to a predefined value (4000). Since the length of the content of each file is completely different, the use of padding was necessary. This was performed by inserting a sufficient number of zeros at the end of the array to be divisible by the predefined width of the image.

After that, there is only one crucial step left to generate images from the data source. Namely, since our data was incomplete in some places, so that it did not contain a real number but a "??" symbol instead, I arbitrarily set this value to 0.

The obtained file was then scaled into a standard image size using the Pillow library.

I applied the following image sizes to train the classifier model: 64x64, 128x128 and 256x256. Although many studies have also considered the analysis of the 32x32 resolution, I considered during the transformation that this size is too small to provide a relevant amount of information for pattern search.

To organize the already generated images, I used a "sort" function I created, which sorted the images according to the train labels in the excel spreadsheet and saved each image in the subfolder corresponding to the train label. Now everything was ready to build the model.

7.1.3 Structure of the model

I have used the Tensorflow library for the final design of the model.[21] As the initial step, I had to convert my already processed data into a structured dataset for which I applied the built-in image_dataset_from_directory function.

The input images were divided into two subgroups. One is the train dataset, which contains 80% of the total data, i.e. 8695 pieces of malware. The other is the validation dataset, which contains the remaining 20%, i.e. 2173 pieces of malware. The validation dataset will allow me to test the accuracy of my classification algorithm in the future.

For the classification I was working with RGB values, i.e. the colour values were made up of 24 bits. Although this could be considered unnecessary, since I used grayscale images, it did not affect the computational complexity to any appreciable extent for such a large amount of data.

Since the size of my dataset was negligible for the amount of memory in the computer, I cached both the train dataset and the validation dataset into the memory, thereby speeding up the classification process.

I implemented a sequential model for the model design. In the first layer, since the pixel values of each image are numbers between 0-255, I had to normalize this to yield a value between 0-1. I did this normalization process using the Rescaling layer. This divided the values for each pixel by 255 to produce the expected input values.

In the following step, I combined a three-step group of one conv2D and one MaxPooling2D layer per step. The composition of each step was as follows:

1. A 3x3 conv2D layer with 16 filter outputs and a 2x2 MaxPooling2D layer
2. In the second step I used a 3x3 conv2D layer with 32 filter outputs, and also a 2x2 MaxPooling2D layer
3. In the third phase, a conv2D with 64 filter outputs and a 2x2 MaxPooling2D were added

The resulting values were then flattened into a one-dimensional vector, and a Dense layer was used to connect the values to a layer of 128 nodes.

Finally, I used a fully connected Dense layer to produce the expected output of nine groups. In the model, the ReLU activation function was used everywhere except for the fully connected layer, where I used the SoftMax activation function.

In the process of compiling my model, I chose the "adam" optimizer as the optimizer and the "loss" values were obtained using sparse categorical crossentropy.

The structure of the resulting model is shown in figure 7, in which the number of trainable parameters is 549,161.

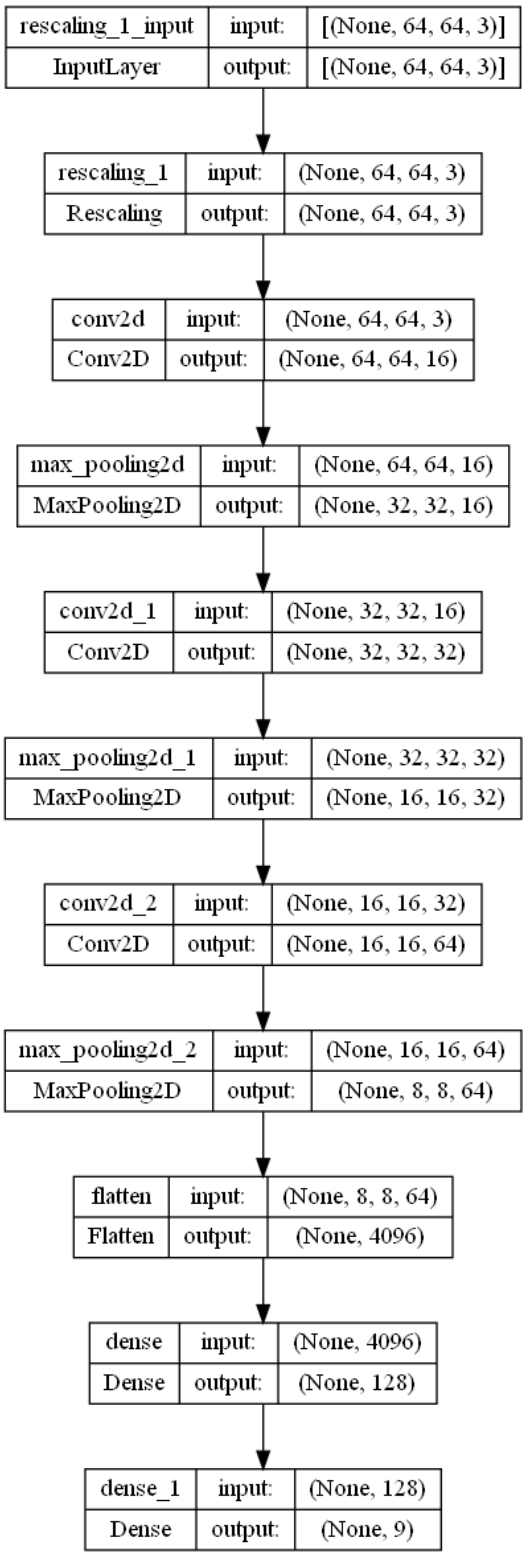

Figure 7: Structure of the image-based model

The dataset cannot be considered completely uniform, since some malware is under-represented. To compensate for this - in order to allow the model to produce more accurate results during learning - I have determined class weights, which the model, during its execution, takes into account.

The model received the input images in batches of 32 sizes and ran through 50 epochs. The execution was tested on both CPU and GPU, with results showing a huge difference in computation time.

7.1.4 Results

On the CPU it took 21 seconds to execute an epoch, while on the GPU it took only 2 seconds to calculate the same. The final highest accuracy achieved was 97.70%.

Figure 8 shows a confusion matrix detailing the accuracy with which the algorithm identified each malware family. It also shows which malware families were confused with which other malware families in the event of a failure.

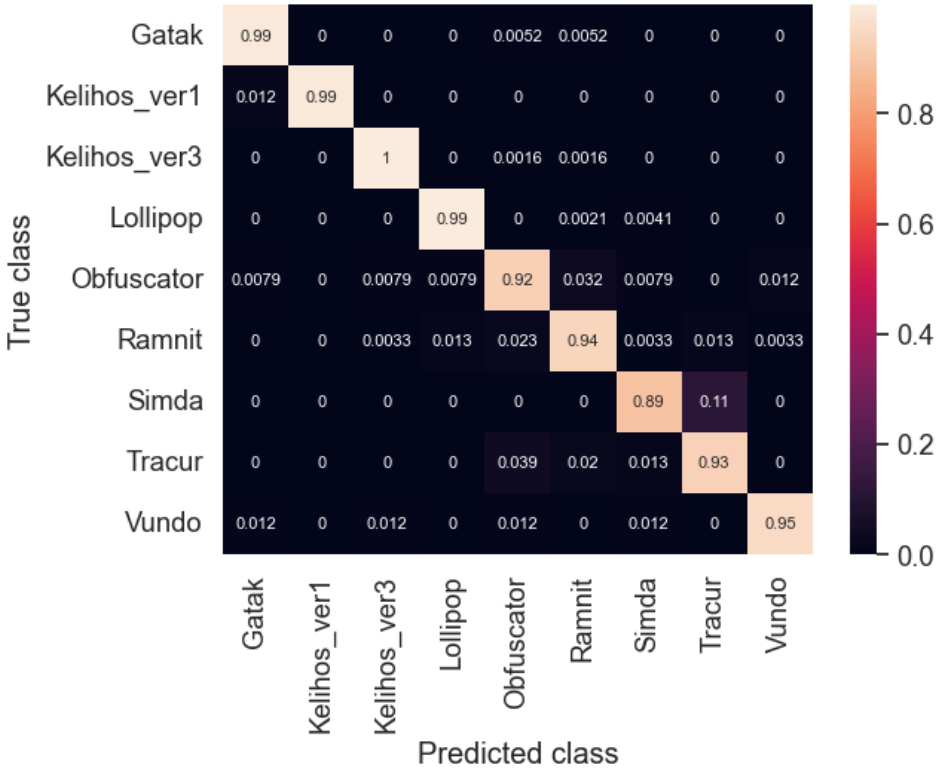

Figure 8: Confusion matrix of the image-based model, with the rows normalized

The diagrams on figure 9 provide information on the accuracy and loss of training and validation at each epoch during learning.

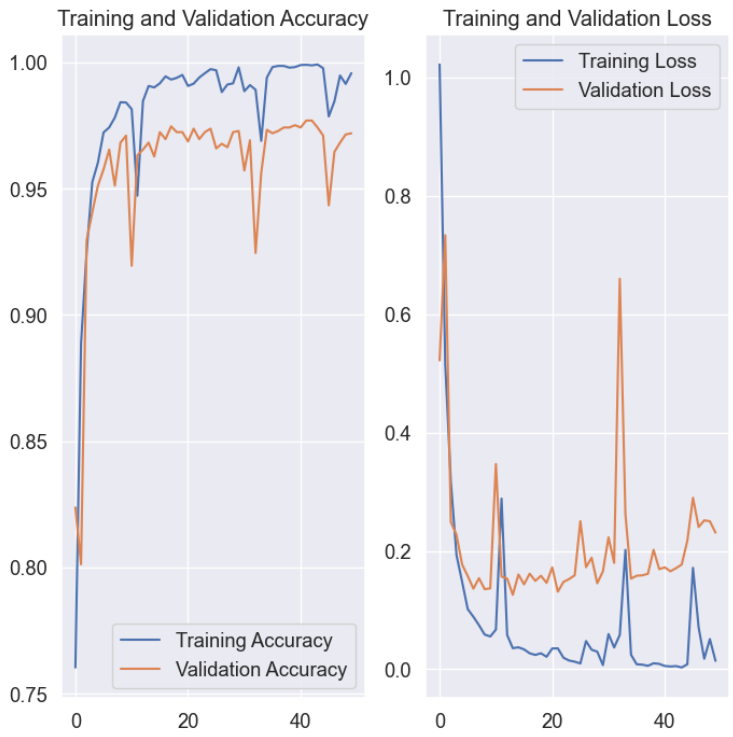

Figure 9: Accuracy and Loss graph of the image-based model

7.2 The Byte-Code based classification model

7.2.1 Specific features and brief description of the model

There are not many available references for direct byte-code-based malware classification on the Internet. There are perhaps several reasons for this, but probably one simple explanation is that it is somewhat more difficult to implement and has more potential pitfalls than the image-based solution described above.

As mentioned earlier, the length of each byte file is different, so choosing the right length can be a crucial factor in achieving maximum accuracy. However, this is not a trivial decision, as the more rows of data we analyze, the more the computational capacity requirement increases. This balance can only be achieved after a long period of experimentation, making human time a key factor in the development of the method.

Furthermore, the information from the limited number of sources reveals that the effectiveness of the method is not outstanding, so this may also diminish its popularity. Nevertheless, in this thesis I have implemented such a model. The exact steps and results are explained below.

7.2.2 Processing the raw dataset

The initial files were the same files described in more detail in the image-based classification model. In this case I have used the bytes files containing the byte codes to train my model.

However, the contents of the files were not usable one-to-one, so I had to reformat them into a format that I could later convert into a dataset. The steps for the conversion are presented below.

As a first step, the row identifier values have been deleted from the file so that they do not affect the outcome of the result.

As a second step, I converted hexadecimal bytes to decimal values in a similar way as in the image-based model. This step was done to make the contents of the two datasets somewhat similar.

Unlike in the image-based model, I did not replace the 'incorrect' parts of the file with a value of 0, but deleted them completely. This way, I have kept only the relevant bytes. After removing the question marks, I compressed the contents of the file, which means that I pushed the first byte with a relevant value into the deleted positions, thus shortening the contents of each file.

Since the average length of such a bytes file is about 80-110 thousand lines, I could not use its entire contents in the classification, as this would have increased the computational complexity enormously and would have resulted in a very slow execution time.

So here I have chosen an arbitrary value for the length. The value in this case was again 4000, already known from the image-based model. In this step I took the first 4000 lines of the bytes file and wrote them to a file with the extension .txt. The generated txt files were given the names of the associated bytes files according to the convention.

Finally, before building the model, I grouped the txt files into the appropriate classes using the tarin labels and the "sort" function mentioned above.

7.2.3 Structure of the model

When creating the model, I used the Tensorflow library as a starting point again.[21]

I converted the processed txt files into datasets using the predefined tensorflow.keras.utils.text_dataset_from_directory function that is provided by Tensorflow.

For the obtained dataset, I applied a split of 80-20%. This means that 80% of the data, i.e. 8695 pieces of malware, are part of the train dataset, while the remaining 20%, i.e. 2173 pieces of malware, are part of the validation dataset.

The elements of the datasets were then vectorised, which is the process of converting each piece of data to be converted into a one-dimensional tensor. It assigns a token to each value of the tensor, which is then used by the model for learning during classification.

To build my model, I used a Sequential model with the following layers:

- The first hidden layer in the model was an Embedding layer, which converts each input word into a vector of fixed length, in this case resulting in the embedding_dim = 64 that I defined. The final vector now has real values, making it easier for the algorithm to interpret each word.
- The second layer is a dropout layer, which works by setting some of the values of the input units to 0 in a completely random way, thus reducing the possibility of overfitting. Its parameter is a float between 0 and 1, which determines the fraction of the input units to be set to 0.
- In the third layer, there is a GlobalAveragePooling1D layer, which takes the average of the feature values and passes it to the next layer.
- This is also followed by a Dropout layer, similar to the one I described above.
- Finally, there are two Dense layers, which are:
  - The first Dense layer binds the input values obtained after the dropout layer with a layer of 128 nodes.
  - While the last one is a fully connected layer, with the output being the nine classes expected in the classification.

The constructed model contains 25,865 trainable parameters. I also used the "adam" optimizer when compiling the model. The loss values were obtained using CategoricalCrossentropy.

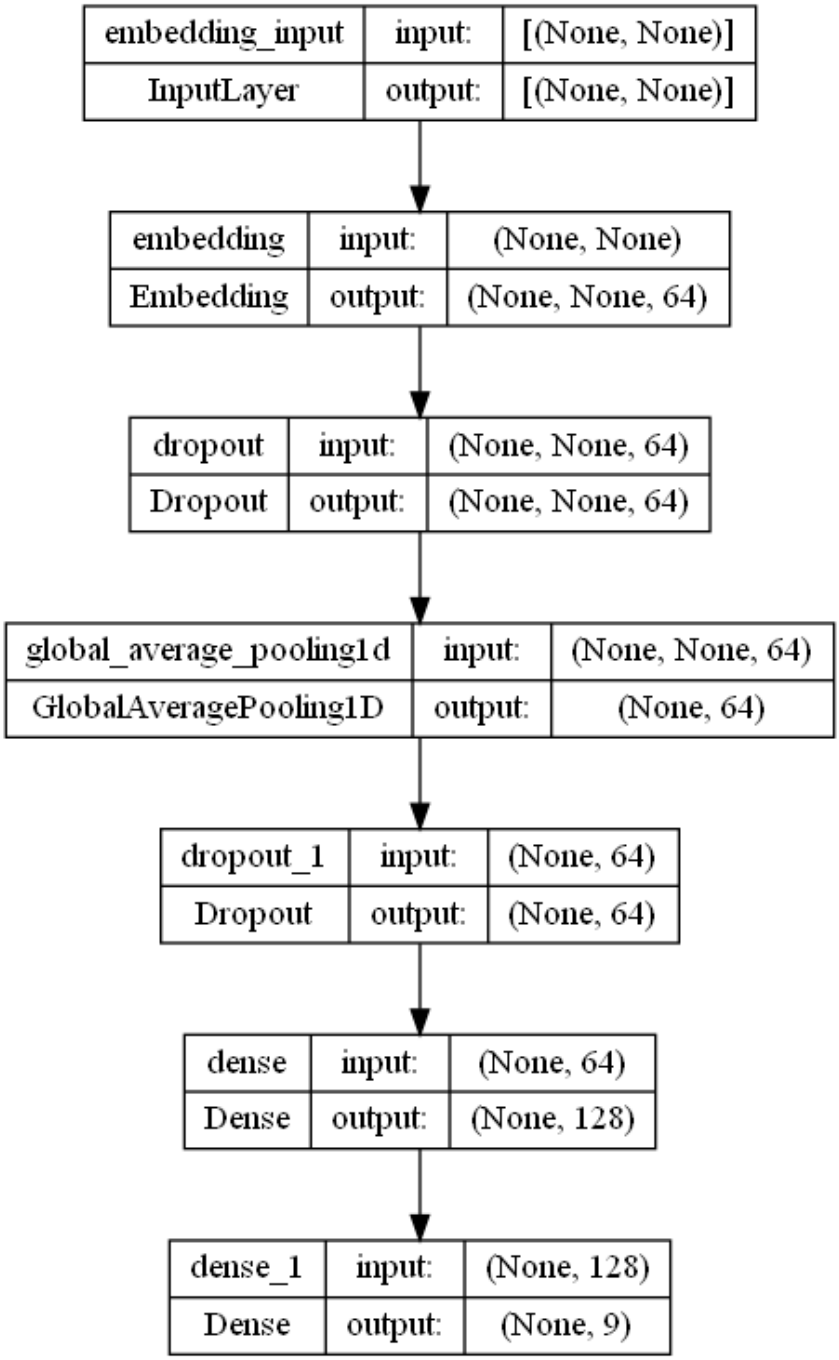

Figure 10: Structure of the byte-code based model

I applied the class weights already described in the image-based model to obtain more accurate results in the categories with lower sample sizes.

The model received the input data in batches of size 32 for the learning process. I chose an epoch size of 50 for the algorithm, which were tested on both CPU and GPU.

7.2.4 Results

Here the gap between CPU and GPU in the execution time of each epoch was much more drastic. It took 321 seconds to execute a single epoch on the CPU, while on the GPU it only took 21-27 seconds.

The highest accuracy achieved by my model was overall 95.67%, underperforming the image-based model.

Figure 11 shows the confusion matrix generated for this model after the learning process.

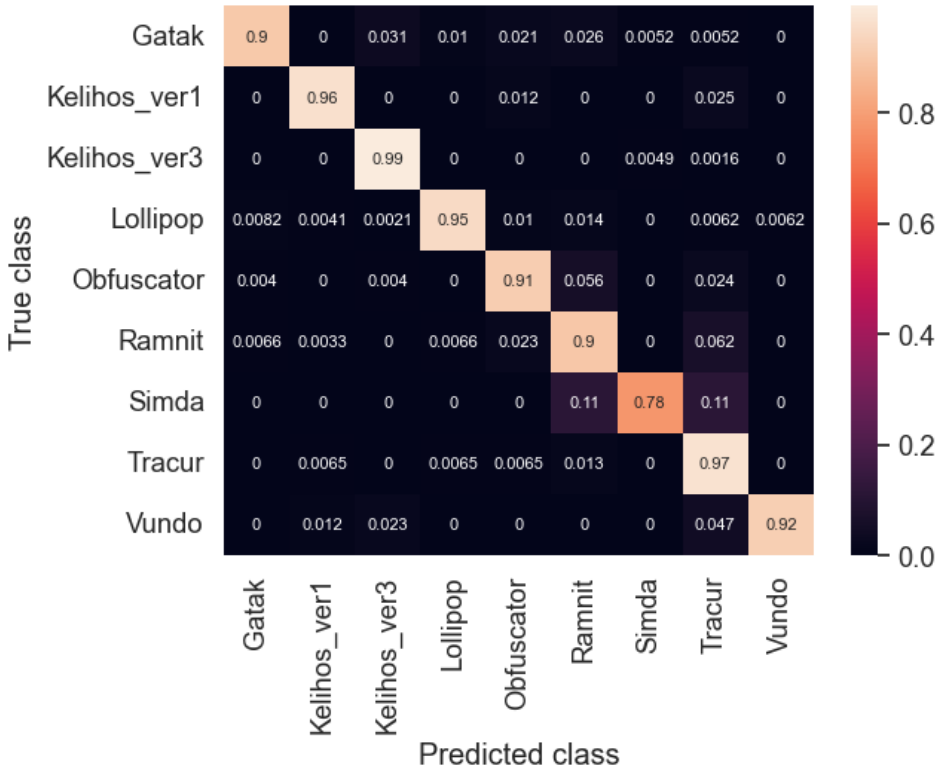

Figure 11: Confusion matrix of the byte-code based model, with the rows normalized

The accuracy and loss achieved while executing each epoch on the train dataset and the validation dataset are illustrated in figure 12.

Figure 12: Accuracy and Loss graph of the byte-code based model

7.3 Introduction to the "mixed" model developed during the study

7.3.1 Specificity and brief description of the model

The "mixed" model I have implemented is based on the image-based model and on the byte-code based model already introduced above. It is named after this, as it combines elements of both approaches.

The inspiration for the model came from the fact that both techniques alone are quite effective at classifying malware, but there is still a small gap for further development.

The individual models could only have been improved with considerable effort, but there is no guarantee of success. Although in this case we also could not be sure that the new method would be more effective than its predecessors, it does provide an interesting insight into an interesting approach.

The "mixed" model owes its effectiveness to the fact that it combines the positive aspects of both techniques described above. The files used in the model, their composition and detailed description, as well as a detailed description of the model itself, are presented below.

7.3.2 Processing the raw dataset

The files used for learning are taken from the files already discussed. No further changes to our dataset were necessary to implement the model.

Here again, I could simply use the images used in the image-based model without any other modifications. I also did not need to make any further changes to the generated txt files, which I could easily use in the model.

This discovery has greatly reduced the amount of time spent on preparation.

7.3.3 Creating the right dataset

In this example, I used the previously tested Tensorflow library to create my model.

However, I encountered difficulties in finding the right dataset. Although it seems to be a reasonable step to build the dataset from the images using the tensorflow.keras.utils.image_dataset_from_directory and from the txt dataset using the tensorflow.keras.utils.text_dataset_from_directory functions, this did not work in this case.

The reason for this was that these functions do not take into account the alphabetical order of each file when building the dataset. As a result, the first element of the generated image dataset, for example, did not match the first element of the dataset generated from the txt file.

To solve this problem, a generator function had to be written. The role of the function is to correct the mismatch between the files.

Since both the image (.png) and text (.txt) folders have the same subfolder (class) names and the same malware files, it was a lot easier to create the function.

Before using the function, one vital step had to be done. This step consisted of creating a random.shuffle shuffled numpy array containing all the malware, with the name of the associated class, without the file extension.

Figure 13 shows an example of a part of the array.

```
['Tracur\\AOEYWM6h02SU5q9dxP7v',
 'Kelihos_ver3\\6E78GVaCMig3mTz1oFb0',
 'Kelihos_ver3\\7BFRaJIPKrv3QVbkDx9C',
 'Tracur\\gVOE7DmPnKrA2Yj5Fz41',
 'Lollipop\\bdE5NOrmaBpsYKGlhIPF',
 'Lollipop\\cARj7FrG15WHEQikbmBU']
```

Figure 13: An example of a part of the array

The principle of the generator function can be described as follows:

- The previously mentioned numpy array is iterated through all its elements using a for loop.
- The currently extracted item is appended so that it contains the full path to its png or txt correspondent of the malware indexed by the item.
- The resulting two paths are used to first load the malware in image format using the tensorflow.keras.utils.load_img helper function, and then store it in an array named train_data_img. In the second round, the txt format malware is loaded and its contents are saved in an array named train_data_txt.
- In the last step of the generator function, the corresponding train labels are defined for each element of the train_data. The given train labels are stored in one hot encoded form in an array called train_data_labels.

7.3.4 Structure of the model

After completing the steps described above, everything was ready to use our dataset to train our model. To do this, however, we still had to assemble the "mixed" model using the appropriate models. The exact description and specification of this is explained below.

The basis for the model can be derived from the two models presented above.

The first is the image-based approach, with the layers in the "mixed" model remained unchanged, except for one layer. The last Dense layer has been outsourced to a Dense layer of 16 nodes, so its output does not consist of the usual nine classes.

The other model on which it is based is the byte-code based approach, whose model is also unchanged in the "mixed" model, with the same exception as for the image-based model.

The obtained models were then concatenated and the structure of the model was deepened using further layers. The structure of the additional layers is as follows:

- First, a Dense layer of 32 nodes combines the output of the image-based and byte-code based models. For this layer, I chose the sigmoid activation function
- In the second step, a dropout layer is added, which I have already described in detail in the byte-code based model.
- Finally, two more Dense layers follow the previous layers, one is a layer of 16 nodes, while the last one is a fully connected layer with the expected output of nine classes.

The optimizer set for the model remains the "adam" optimizer. The losses were obtained using CategoricalCrossentropy. In the end, the composed model consists of 20 layers and contains 578,281 trainable parameters.

As an additional optimization option, I have also used the class weights discussed above for the "mixed" model too. The training of the model is illustrated in the following code fragment:

```
history = model_Mixed.fit(x=[train_data_txt, train_data_img],
                          y=train_data_labels,
                          class_weight=class_weights_dict,
                          validation_data=[(val_data_txt,
                                            val_data_img),
                          val_data_labels], epochs=epochs)
```

For the model, I set the epoch size to 50 and tested its execution on both CPU and GPU.

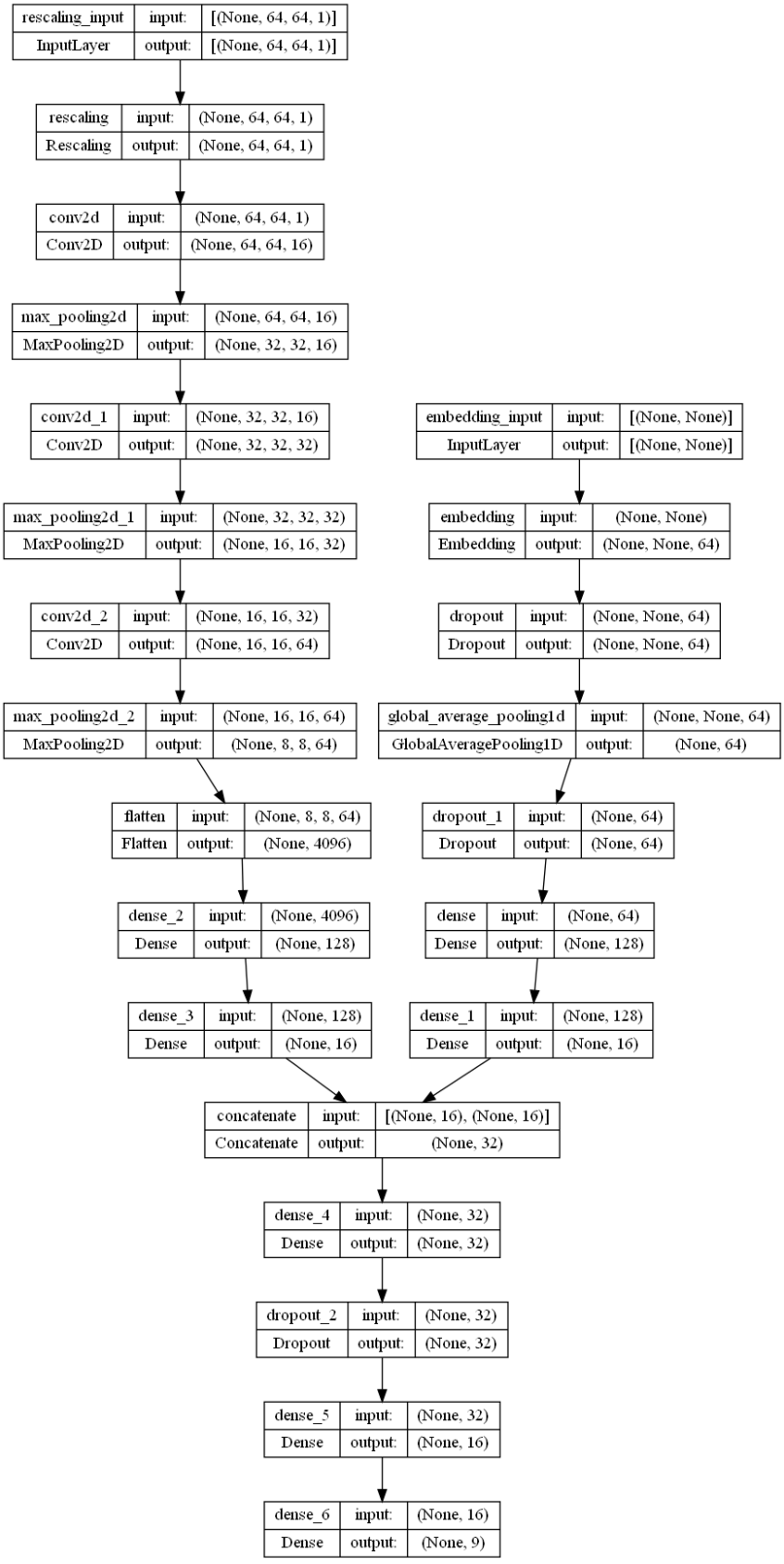

Figure 14: Structure of the mixed model

7.3.5 Results

There were also significant differences in the time needed to complete each epoch in the "mixed" model. When executed on the CPU, it took 327 seconds to process a single epoch. The GPU shortened this time requirement by a large margin, taking only 21 seconds.

The highest accuracy achieved with the model was 98.42%, outperforming all previously presented solutions.

Figure 15 shows the confusion matrix for the "mixed" model.

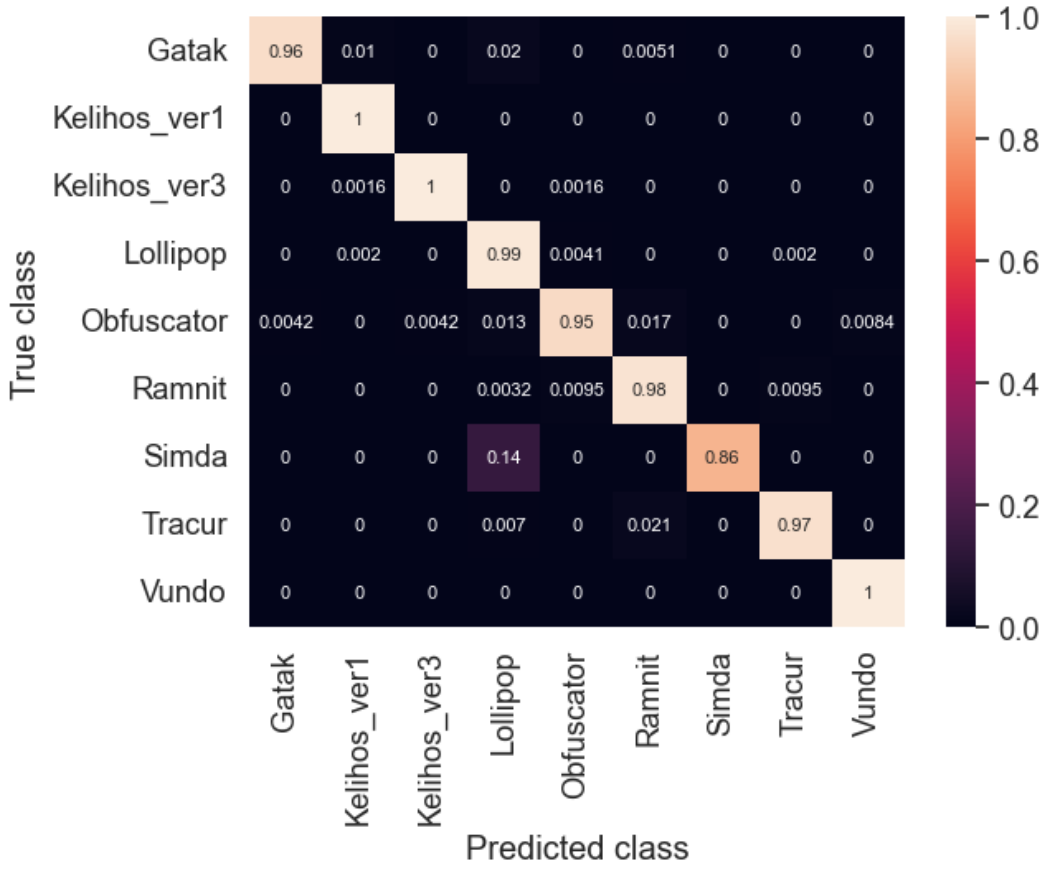

Figure 15: Confusion matrix of the mixed model, with the rows normalized

The accuracy and loss values obtained on the train data and validation data during each epoch are illustrated in figure 16.

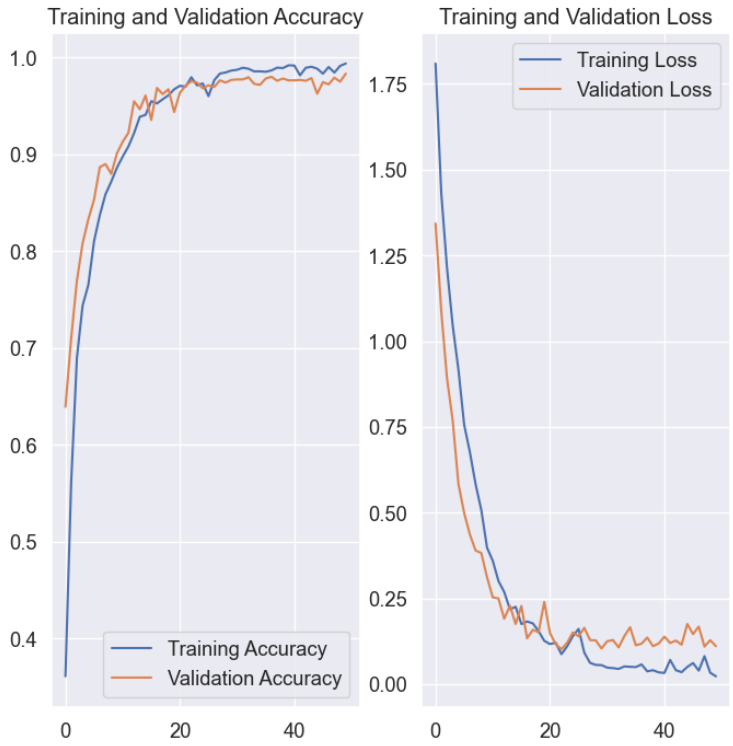

Figure 16: Accuracy and Loss graph of the mixed model

8 Comparing and analysing the results obtained by the models

In this chapter, the results achieved by each model are presented and compared. For the purpose of completeness, the results obtained are also included below:

By implementing the worst performing byte-code based model, I was able to classify each malware with an accuracy of 95.67%.

With the image-based model, this accuracy rate was increased to 97.70%, outperforming the byte-code based model.

The model with the highest accuracy level was the "mixed" model, which achieved 98.42%.

This shows that of the three different approaches, the "mixed" model performed best in terms of accuracy. However, to say that this is the most effective model for classifying malwares seems a bit premature. In order to do so, it is also worth considering the times required to execute the models.

In terms of execution times, the image-based model presented first produced the shortest time consumed. It took only 2 seconds on the GPU. The remaining two models when they were executed on GPU produced nearly identical times (21-22 seconds) with a few seconds difference. However, this still represents a significant 19-20 seconds overhead.

model name	avg accuracy on validation dataset (%)	best accuracy on validation dataset (%)	execution time/epoch on GPU in seconds
image-based	96,78	97,71	1
byte-code based	93,17	95,67	22
mixed	95,38	98,42	21

Figure 17: Comparison between the models

8.1 Conclusion

Nevertheless, we can say that the accuracy of the best "mixed" model that has been created for this paper shows an improvement of 0.72% compared to the best performing models that I have presented. Although it may be considered a minor improvement numerically, however in the case of a computer infrastructure aiming for high security, every tiny percentage may have a huge impact.

On the other hand, we can also consider that in some cases, where the available resources are more limited and the data stored is less critical from a security point of view, a simple image-based model may be the most appropriate way to detect malwares.

8.2 Comparison of the "mixed" model with examples in the literature

Analysing existing examples in the literature, accuracy rate varies widely across models. The average ranged from 91-95% [8][9]. Considering only the image-based approach, the average ranged from 96-98% [19], but there was also an extremely outstanding value of 99.7% [1].

Based on this information, it can be stated that the byte-code based approach I have constructed has an average performance compared to its counterparts in the literature. Furthermore, my image-based model met, and even slightly exceeded, the average values of the other image-based models.

The innovative "mixed" model outperformed the average, putting it in a leading position in the competition. But there are also models with even higher accuracy rates. Thus, the "mixed" model can compete with its competitors, but it is not the best.

8.3 Future works

Here are some useful ideas for further improvements and a new approach to the model.

In terms of further development of the model, with further optimisation steps it would be worthwhile to reduce the computational cost required to execute the model. This would decrease the deviation compared to image-based models. This might also make it a better choice for systems with fewer resources.

Additional changes could be made to try to increase the accuracy achieved by the "mixed" model. It might also be a good idea to change the composition of the two underlying (base) models. However, this has not been done in this paper in order to obtain accurate benchmarks.

The "mixed" model could be further extended with a new approach, by extracting process dumps of executable malwares and feeding them back into the model using a similar hybrid framework approach as presented above.

This can be done in several ways, either by returning the process dump using only either the byte-code based or only the image-based approach. But implementing both may be a good option to further increase accuracy. This way we get a dataset of four different data sources, on which we build four different models and join them in the "mixed" model.

9 Acknowledgements

I would like to express my appreciation to Dr. Gábor Hullám and Mihály Vetró for their help and advice during the preparation of the TDK thesis and the related background work.

References

[1] Rajasekhar Chaganti, Vinayakumar Ravi, and Tuan D. Pham. "Image-based malware representation approach with EfficientNet convolutional neural networks for effective malware classification". In: *Journal of Information Security and Applications* 69 (2022), p. 103306. ISSN: 2214-2126. DOI: https://doi.org/10.1016/j.jisa.2022.103306. URL: https://www.sciencedirect.com/science/article/pii/S2214212622001570.

[2] Dettmers, Tim. *Deep Learning in a Nutshell: History and Training*. https://developer.nvidia.com/blog/deep-learning-nutshell-history-training/. 2015.

[3] Bin Ding, Huimin Qian, and Jun Zhou. "Activation functions and their characteristics in deep neural networks". In: *2018 Chinese Control And Decision Conference (CCDC)*. 2018, pp. 1836–1841. DOI: 10.1109/CCDC.2018.8407425.

[4] Levente Imre Dobák. *Source Code*. https://github.com/entwickler-dli/tdkmalwareclassification.

[5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[6] Kaivan Kamali. *Deep Learning (Part 3) - Convolutional neural networks (CNN) (Galaxy Training Materials)*. https://training.galaxyproject.org/training-material/topics/statistics/tutorials/CNN/tutorial.html. [Online; accessed Tue Nov 01 2022].

[7] Anup Kumar and Alireza Khanteymoori. *Introduction to deep learning (Galaxy Training Materials)*. https://training.galaxyproject.org/training-material/topics/statistics/tutorials/intro_deep_learning/tutorial.html. [Online; accessed Tue Nov 01 2022].

[8] Sanjeev Kumar and B. Janet. "DTMIC: Deep transfer learning for malware image classification". In: *Journal of Information Security and Applications* 64 (2022), p. 103063. ISSN: 2214-2126. DOI: https://doi.org/10.1016/j.jisa.2021.103063. URL: https://www.sciencedirect.com/science/article/pii/S2214212621002465.

[9] Lin Li et al. "Malware classification based on double byte feature encoding". In: *Alexandria Engineering Journal* 61.1 (2022), pp. 91–99. ISSN: 1110-0168. DOI: https://doi.org/10.1016/j.aej.2021.04.076. URL: https://www.sciencedirect.com/science/article/pii/S1110016821003185.

[10] Wei-Cheng Lin and Yi-Ren Yeh. "Efficient Malware Classification by Binary Sequences with One-Dimensional Convolutional Neural Networks". In: *Mathematics* 10.4 (2022). ISSN: 2227-7390. DOI: 10.3390/math10040608. URL: https://www.mdpi.com/2227-7390/10/4/608.

[11] *Malware types*. https://www.malwarebytes.com/.

[12] NIST. *Malware Definition*. https://www.nist.gov/.

[13] Stephen O'Shaughnessy and Stephen Sheridan. "Image-based malware classification hybrid framework based on space-filling curves". In: *Computers & Security* 116 (2022), p. 102660. ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2022.102660. URL: https://www.sciencedirect.com/science/article/pii/S0167404822000591.

[14] *Picture of Alexey G. Ivakhnenko*. https://en.wikipedia.org/wiki/File:Photo_of_Prof._Alexey_G._Ivakhnenko.jpg.

[15] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. *Searching for Activation Functions*. https://arxiv.org/abs/1710.05941. 2017. DOI: 10.48550/ARXIV.1710.05941.

[16] Royi Ronen et al. *Microsoft Malware Classification Challenge*. 2018. DOI: 10.48550/ARXIV.1802.10135. URL: https://arxiv.org/abs/1802.10135.

[17] Sagar Sharma, Simone Sharma, and Anidhya Athaiya. "Activation functions in neural networks". In: *towards data science* 6.12 (2017), pp. 310–316.

[18] P Sibi, S Allwyn Jones, and P Siddarth. "Analysis of different activation functions using back propagation neural networks". In: *Journal of theoretical and applied information technology* 47.3 (2013), pp. 1264–1268.

[19] Tran The Son et al. "An enhancement for image-based malware classification using machine learning with low dimension normalized input images". In: *Journal of Information Security and Applications* 69 (2022), p. 103308. ISSN: 2214-2126. DOI: https://doi.org/10.1016/j.jisa.2022.103308. URL: https://www.sciencedirect.com/science/article/pii/S2214212622001594.

[20] Rabia Tahir. "A study on malware and malware detection techniques". In: *International Journal of Education and Management Engineering* 8.2 (2018), p. 20.

[21] *Tensorflow Framework*. https://www.tensorflow.org/.

[22] Ilsun You and Kangbin Yim. "Malware Obfuscation Techniques: A Brief Survey". In: *2010 International Conference on Broadband, Wireless Computing, Communication and Applications*. 2010, pp. 297–300. DOI: 10.1109/BWCCA.2010.85.

[23] Fangtian Zhong et al. "Malware-on-the-Brain: Illuminating Malware Byte Codes with Images for Malware Classification". In: *IEEE Transactions on Computers* (2022), pp. 1–1. DOI: 10.1109/TC.2022.3160357.