FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Malware Classification Based on Self-Organizing Maps
Student:	Bc. Vojtěch Skalák
Supervisor:	Mgr. Olha Jurečková
Study program:	Informatics
Branch / specialization:	Computer Security
Department:	Department of Information Security
Validity:	until the end of summer semester 2023/2024

Instructions

Self-organizing maps (SOM) is an unsupervised machine learning technique that transforms a complex high-dimensional input space into a simpler low-dimensional. The SOM have found applications in intrusion detection systems that monitor network traffic. Still, their use for classifying malicious binaries is not yet well explored. This work aims to investigate the possibilities of using self-organizing maps for the malware classification problem, specifically under the Windows operating system.

Instructions:
1. Study the existing approaches to malware classification procedures/techniques and review previous research based on self-organizing maps.
2. Implement self-organizing maps.
3. Choose an appropriate publicly available dataset and train a self-organizing map to classify malicious samples.
4. Discuss the results and compare them with other malware classification techniques.

Electronically approved by prof. Ing. Róbert Lórencz, CSc. on 26 January 2023 in Prague.

Master's thesis

MALWARE CLASSIFICATION BASED ON SELF-ORGANIZING MAPS

Bc. Vojtěch Skalák

Faculty of Information Technology
Katedra informační bezpečnosti
Supervisor: Mgr. Olha Jurečková
June 28, 2023

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis: Skalák Vojtěch. *Malware Classification Based on Self-Organizing Maps.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Contents

List of Figures

List of Tables

List of code listings

I would like to thank my advisor, Mgr. Olha Jurečková, for her invaluable guidance and support.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis. I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. I further declare that I have entered into a licence agreement with the Czech Technical University in Prague for the utilization of this thesis as a school work pursuant to Section 60(1) of the Copyright Act. This fact does not affect the provisions of Section 47b of Act No. 111/1998 Coll., the Higher Education Act, as amended.

In Praze on June 28, 2023 .....

Abstract

This paper examines malware classification using self-organizing maps. Detection using artificial intelligence is one way to effectively combat the spread of malware, and the ability of self-organizing maps to organize data into clusters can aid in classification. This paper summarizes the theoretical basis and the development of self-organization maps so far. Afterward, a test map is trained on a publicly available dataset with parameters selected for this type of data. In the last section, the measured results are compared with other machine learning methods.

Keywords Self-organizing map, Kohonen map, Machine learning, Malware classification

Abstrakt

Tato práce se zabývá klasifikací malware pomocí samoorganizačních map. Detekce pomocí umělé inteligence je jedním ze způsobů, jak efektivně bojovat proti šíření malware a schopnost samoorganizačních map uspořádat data do klastrů mohou pomoci při klasifikaci. V práci je shrnut teoretický základ a dosavadní vývoj samoorganizačních map. Posléze je na veřejně dostupném datasetu natrénována zkušební mapa s parametry vybranými pro tento typ dat. V poslední části jsou naměřené výsledky porovnány s ostatními metodami strojového učení.

Klíčová slova Samoorganizační mapy, Kohonenovy mapy, Strojové učení, Klasifikace malware

Summary

Introduction

The theoretical basis of self-organizing maps and the mathematical principles behind some of the functions used.

Literature review

The summary of the development in the self-organizing maps, malware classification and machine learning

Methodology

The methods used for data and parameter selection.

Results

Results achieved by the self-organizing map on the selected dataset and results of the other machine learning methods.

Discussion

Comparison of the results and discussion.

Abbreviations

API	Application Programming Interface
ML	Machine learning
PE	Portable Executable
DLL	dynamic link libraries
SOM	Self-organizing map
BMU	Best-matching unit
PUP	Potentially unwanted program
QE	Quantization error
TE	Topographic error

Introduction

Malware classification

Malware is a combination of the words malicious and software. It is software whose purpose is to harm either user or the computer. To classify malware, the terms type and family are used. The malware type refers mainly to its behavior. There are many types of malware, the most common include Trojans, viruses, spyware, or worms. The malware family refers to characteristic features of the malware and is used to mark specific groups or variants.

The malware classification techniques are used to divide malware samples into families. Assigning a family to a malware sample provides information about the malware's behavior and characteristic features. Furthermore, it improves malware signature generation, one of the malware detection techniques. [1]

Malware detection

Malware detection techniques aim to decide, whether given software is malware or not. As malware is continuously evolving and poses a threat as the number of both attacks and devices in the world is increasing, the need to rapidly and effectively detect malware samples is rising. Also, detecting malware becomes more challenging with the number of malware types increasing. There are several approaches used to detect malware:

The signature-based detection consists in reverse-engineering malware, extracting patterns, and selecting labels based on the pattern. Detection on endpoints is done by comparing hashes with a vast antivirus database stored either at the endpoint itself or in the cloud so that it can accommodate a more extensive amount of data. This approach is fast and effective, however, it may be insufficient when detecting new types or variants of malware.

The behavior-based detection is determining if the software is malicious based on its behavior. It is using various tools to capture API calls, system calls, etc. Contrary to the signature-based approach, it can detect new types of malware if their behavior is similar.

The heuristic-based detection uses rules and ML techniques to achieve a high accuracy rate of malware detection on zero-day, low-complex malware. However, it is not efficient at detecting malware that is using metamorphic techniques.

Model checking-based detection , used for system verification, creating a formal specification of software given using mathematical language and logical formulas. The model created either manually or automatically is then checked for any behavior matching known malware.

This technique is effective against obfuscated or polymorphic malware. The main disadvantage is that this approach is computationally expensive, especially for more complex systems.

Deep learning-based detection learns from malware samples. It is an effective detecting method but it is also vulnerable to evasion attacks i.e. manipulation of the input data aimed to confuse the machine learning model.

From those detection techniques, the highest accuracy rate when detecting more complex malware achieve behavior-based methods, while the signature-based technique has the lowest accuracy rate. [2]

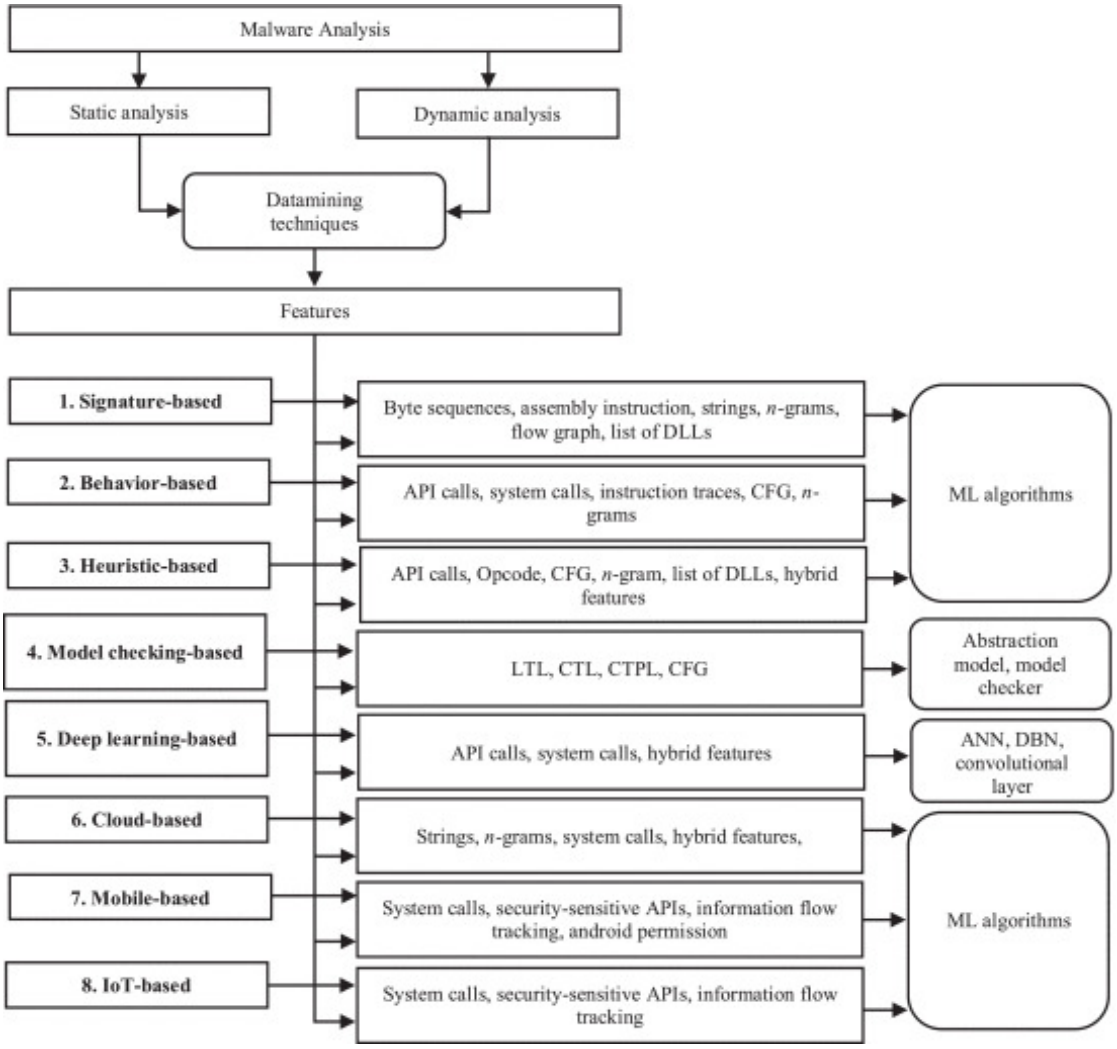

Figure 1 Malware detection techniques overview [2]

The malware detection techniques can be improved by malware classification techniques and together lead to a faster and more efficient detection method. An overview of all the techniques can be seen in figure 1

PE file format

The Portable Executable (PE) file format is a binary format used in the Microsoft Windows OS. It is used for executable files, dynamic link libraries (DLL), device drivers, or object code files. Each PE file includes a header with several sections. The overview of the section can be seen in 2. Brief contents of the sections:

`DOS Header` An executable header allowing the file to be executed in MS-DOS. Every PE file starts with this header.

`PE Signature` Fixed 4-byte signature ("PE
0
0") marking the beginning of the PE file.

`COFF Header` (Common Object File Format Header) Contains information about the executable such as architecture type, number of sections, timestamp, entry point, etc.

`Optional Header` Additional information about the executable.

`Section Header` Information about the sections of the PE file.

Figure 2 The PE file header

Self-organizing map

A self-organizing map (referred to as 'SOM'), also called Kohonen or feature map, was first described by Teuvo Kohonen in [3]. According to [4] and [5], it is a neural network architecture that is used to represent high-dimensional input data in a low-dimensional format. To train the network unsupervised, competitive learning is used.

Unsupervised learning is the training of neural networks on unlabeled data sets. The network's output is therefore not compared with the expected output; instead, the network self-organizes its neurons. This leads to data clustering based on input data features. Competitive learning means that the neurons compete for each input vector based on a selected metric. Only one winning neuron is selected for each input, although more than one can be updated.

Each neuron has a weight vector initialized to a value that is:

1. random
2. picked from input space
3. picked to reflect the distribution of the input space

The third option is preceded by Principal Component Analysis (referred to as 'PCA'). PCA is also used to reduce the dimension of the data set without significant loss of information. According to [6] random initialization is seldom used for practical purposes because it lowers the convergence rate of the learning process. Values for initialization may be derived from the two largest principal components so that the values reflect the distribution of the input space.

These initiated vectors, called models, are associated with neurons, which are organized into a grid. For most cases, [6] recommends two dimensional, hexagonal-shaped grid. A higher dimension grid can be used for special purposes only as one of the features of SOM is a representation of high-dimensional data into a low-dimensional plane. Toroidal or spherical grids may be used for the data representation across grid borders to be smoothed.

Implementation of the SOM network can follow two different approaches. The first approach is a step-wise algorithm that processes one input vector at a time. The main idea of the SOM algorithm is as follows. Input data item selected at random or in predefined order is compared with all models associated with all neurons of the network. The winning model is determined based on a function, most commonly Euclidean distance. This is also called the competitive process [4] as the neurons compete with each other for the input. After selecting the winning neuron or Best Matching Unit (referred to as BMU), several neurons that are spatially near him on the grid are selected based on the neighborhood function (1). This step of the algorithm is called the cooperative process. In the next step, all selected neurons are updated by function (2), also called the adjustment process.

According to [6] function (1) is often used as a neighborhood function.

Example of neighborhood function choice:

$$h_{ci}(t) = \alpha(t) exp[(-sqdist(c,i)/2\sigma^2(t))] \tag{1}$$

Indexes c and i again represent neurons for which the neighborhood distance is calculated, c is the winning neuron and i is the neuron on which the function is applied. Function $\alpha(t)$ is a monotonically decreasing scalar function of t and $\sigma(t)$ is a monotonically decreasing function of t. The function $sqdist(c,i)$ is a squared distance between neurons c and i. This function with minor changes is also proposed by [5] as a neighborhood function.

The update function is defined as follows:

$$\vec{\omega}_{\vec{s}}(t+1) = \vec{\omega}_{\vec{s}}(t) + \epsilon h(\vec{r},\vec{s})(\vec{v}(t) - \vec{\omega}_{\vec{s}}(t)) \tag{2}$$

The update function (2) as described in [7] contains neighborhood function $h(\vec{r},\vec{s})$ so the function is applied to all the neurons after each input. Vector $\vec{r}$ represents the position of BMU

and vector $\vec{s}$ represents the position of the neuron on which the update is applied. Function $h()$ represents a function that decreases with the distance between the two nodes. Distance is calculated from their coordinates in the network topology. Vector $\vec{\omega}_{\vec{s}}$ denotes the weight vector associated with neuron s, the one that is updated by the function. Symbol ϵ represents the learning step. It is a constant set as $0 < \epsilon < 1$. The nodes that do not belong to the neighborhood distance are therefore not affected.

Euclidean distance is the most common choice according to [4], but other metrics can be used as well. In the two-dimensional space of the SOM, the Euclidean distance between two points $A = (x_1, y_2)$ and $B = (x_2, y_2)$ is defined as follows:

$$d(A, B) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

This metric calculates the direct distance between two points and in the context of the SOM, it preserves the relative distances. However, it could be influenced by extreme values more easily than the other metrics described below.

Another option is to use the Manhattan distance, which is a measure of distance in a lattice structure. The distance is calculated as:

$$d(A, B) = |x_2 - x_1| + |y_2 - y_1|$$

The Manhattan distance is commonly used if the points are distributed in a grid-like structure. It is less influenced by extreme values than the Euclidean distance. However, it may not reflect the shortest path between the points. It is a common choice for a structure with points distributed over two perpendicular directions.

The third metric discussed in this work is the Chebyshev distance. Like the Manhattan distance, it is measured on a lattice structure, but it is calculated as the maximum difference between two points' coordinates along any dimension:

$$d(A, B) = max(|x_2 - x_1|, |y_2 - y_1|)$$

This metric is an alternative to Manhattan distance on grid-like structures. It is even less influenced by extreme values, as can be derived from the metric equation, and it is a more suitable choice to be used in a lattice structure with more than two directions, i. e. measure the distance between points on a diagonal.

After the three steps of the step-wise approach, the stopping condition is checked. The algorithm can stop if the network reached a stable state. In such a state, the values of models do not change in time: for each model m_i associated with neuron i the following equation is satisfied:

$$m_i(t + 1) = m_i(t) \tag{3}$$

From equation (3) where $t \to \infty$ the equation for batch computation (4) is derived.

An example of a step-wise learning process in a two-dimensional hexagonal-shaped grid can be seen in Figure 3. Label A denotes the data item from input space which is compared with all models, B is labeled the BMU, the winning neuron, while C is the neighborhood at some point in the learning process. Models associated with all of the neurons in area C are updated in this iteration.

The second approach to implementing SOM is the batch-type process. With the presumption of a finite number of input data items, all of these items are applied in one batch. If there is a very large number of input items, a selected subset is applied instead. Models of neurons in this approach are not updated one by one but all at once. For each model m_i are selected data items that are closest to the model, that is m_i is BMU for data items assigned to the associated neuron. The neighborhood is specified for neurons and new values of models are computed from means of values across the whole neighborhood using the function (4) derived from stable state equation (3).

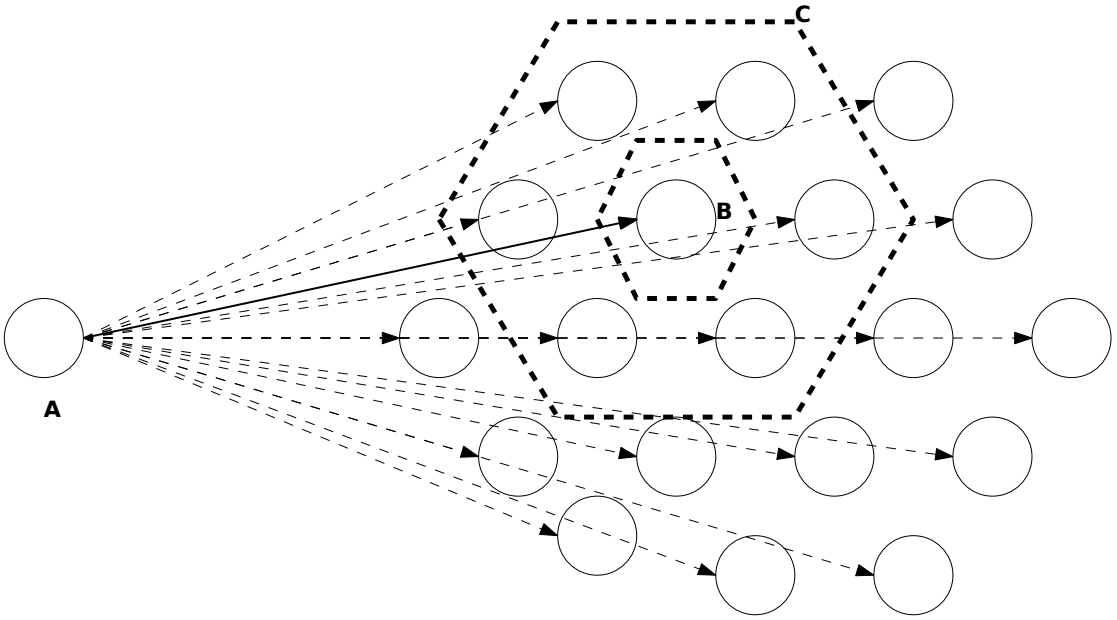

Figure 3 Step-wise learning in hexagonal SOM

Batch-computation update function:

$$\mathbf{m}_\mathbf{i}^* = \frac{\sum_j n_j h_{ji} \mathbf{x}_{m,j}}{\sum_j n_j h_{ji}} \tag{4}$$

The function computes the value of $\mathbf{m}_\mathbf{i}^*$ which is the value of the model associated with neuron i and which is independent of the current time step. For this reason, all input items can be processed at once during one iteration of the batch algorithm. $\mathbf{x}_{m,j}$ represents the mean of inputs assigned to the model m_j in the first step of the iteration. Value n_j denotes the number of these inputs and h_{ji} represents the neighborhood function of neurons m_i and m_j.

This concludes one iteration of the batch processing. Again the process is repeated until a stable state is reached. According to [5] the number of iterations must be at least 500 times the number of neurons in the output layer.

In the general description of the SOM learning process, the learning rate and neighborhood size decrease over time. The changes in these parameters divide the learning process into two phases described in [8]. The phases are:

1. Great topological changes phase
2. Fine-tuning phase

The first phase is denoted by great changes in the topological space of the network. In this phase, the size of the neighborhood and the learning rate are both large. Changes in neuron values and positions are bigger than later. The aim of this phase is approximate the density of input space distribution.

In the second phase, the network does not undergo great topological changes, only small updates of one or a few neurons. The target of this phase is to train the network to distinguish between patterns that are close in the input space but do not belong to the same cluster.

An essential aspect of SOM is network size, which shall reflect the size of the input space. According to [6] the use of SOM on small data sets is not practical, as there are other methods to analyze such data. Furthermore, [9] states that the size of the network affects the initial size of

the neighborhood and the number of iterations. Another characteristic that depends on network size is the granularity of the final feature map. A smaller network size to data space size ratio will result in a coarse texture of the feature map displaying only the most significant clusters. SOM network with a bigger ratio will produce a more precise division of input space features and create more clusters. The drawback of creating a network that is too large is that it may produce sparse data on the output. Again the size of the network can correlate with the two largest principal components of the input data. For these reasons, the network can contain from a few dozen to several hundred neurons. In [6] the author claims that 50 items per neuron are enough for practical usage.

The SOM is a method close to the k-means clustering, which is a vector quantization method. Similarities can also be found with dimensionality reduction methods. In [10] the authors assess the SOM method as a partitioning method with practical usage, although it differs from other partitioning and dimension-reduction methods. The authors conclude that SOM is a useful technique. These are the most useful objectives they assumed after a series of experiments:

1. unsupervised classification method

2. object classification while the object groups are known

3. testing of theoretical models

The authors point out that the method can accomplish these objectives simultaneously. The capability to be practically used as an unsupervised classification method is derived from the fact that SOM is clustering data based on their similarity. The authors also state that the network associates a similar number of items with each cluster center. Classification of objects into known groups can be useful for malware detection and analysis as will be discussed in the next section. Usefulness for testing of theoretical models authors proved by experimenting with real-life objects, stars in this case, which were merged with tracer objects, synthetic objects with properties to match the presumed cluster center. The Kohonen map resulting from applying the SOM method to such data fulfilled the presumptions and divided the map into regions corresponding to the synthetic object's properties.

Chapter 1
Literature review

Potential profits from successful malware are rising and so is the number of new malware types and modifications. In the work [11] authors further claim that malicious actors evolve and adapt their strategies as well as existing malware to be able to exploit vulnerabilities as soon as they are revealed.

New versions of the same malware appear more often. For example in [12] authors state that Agobot, a malware family written in C++, appeared between the years 2002 and 2007 in more than 580 variants.

In the work [11] authors found significant differences between labels issued by various AV providers, as suggested by the comparison of several malware samples of VirusTotal, a cloud solution that compares results of more than 50 AV vendors.

They explained that this is because types of malware are often similar to one another, presenting the same methods and behavior. Thus determining one exact type for each malware is often not possible and the label assigned this way does not have to correspond with the malware's real properties. On the contrary, SOM based detection method processes the input data in such a way, that the resulting feature map not only divides malware into clusters, possibly representing malware families but also shows the similarity between each sample based on their respective position. This feature of SOM that comes from a combination of cooperative and competitive approaches can define a relationship between individual samples better than single labels.

For these reasons, there is a need to implement new ways of malware detection. One of the promising approaches is malware clustering using a neural network.

To analyze malware it is necessary to find its characteristic features. these parameters of malware must differentiate one sample from another. On Windows, one of the common approaches is analyzing API calls as it was used in [13] and [11].

The number of malware grows and new samples appear every day. According to [14] there are more than 450 000 new samples every day. To cope with new malicious programs, [13] proposed a novel approach to classifying malware samples. The main idea is that the new samples are often made by modifying existing malware and thus have the same behavior patterns. These samples can be filtered from those that are new and speed up the detecting process.

To extract malware features, several methods can be used. In the work [2] the authors mention these data mining techniques used for malware features extraction:

The n-gram model is a widely used feature extraction technique that uses both static and dynamic extraction. The technique groups system and API calls in successive order for a given n, creating unigrams for $n = 1$, bigrams for $n = 2$, trigrams for $n = 3$, etc. The disadvantage is that it creates vast feature space. Nevertheless, it is widely used for malware feature extraction.

Graph-based model technique converts system calls into a graph. With increasing complexity, sub-diagrams are created to describe the graph. With few enough nodes and edges, the classification of the software can be done. However, many studies define the creation of the sub-diagrams as NP-Complete and thus computationally very demanding.

Malware dataset prepared to be used in research. Instead of extracting data from malware, one of the well-known and widely used datasets can be used. The authors list the following datasets:

- NSL-KDD dataset (2009) [15]
- Drebin dataset (2014) [16]
- Microsoft malware classification challenge dataset (2015) [16]
- ClaMP (Classification of Malware with PE headers) dataset (2016) [17]
- AAGM dataset (2017) [18]
- EMBER dataset (2018) [19]

Other methods to collect malware features can be used. In [13] the authors used dynamic analysis to extract malware features. They recorded the first 200 API calls plus complementary information. Then they stored it in the matrix and modified it so that the same patterns appeared. For sample classification, supervised learning was used with a total of 42 068 samples, 67 % as a training set and the rest as the testing set. The authors used labels provided by VirusTotal and selected four different types of malware to detect:

1. Trojan
2. Potentially unwanted program (PUP)
3. Adware
4. Rootkit

Overall results were very satisfactory. It worked well for Trojans, PUPs, and Rootkits. Although they measured a significant false positive rate for Adware, the weighted averaged results were not greatly affected and the classifier was evaluated as able to filter out known modified known malware.

In [11] authors based feature selection on API calls of malware and their parameters. API calls were collected during the first two minutes of malware execution. API calls were divided as follows:

1. Passed API calls - managed to successfully change the system state
2. Failed API calls - failed to change the system state
3. Return codes of the failed API calls

This method led to hundreds of features representing malware. To process by SOM, Principal Component analysis had to be used to reduce the number of features. these features were then sent to SOM in form of input vectors. The results of this approach again differentiate from the labels provided by AV vendors.

Another method is to analyze PE files and headers and extract features from them. PE stands for Portable Executable and it is a file format standardized by Microsoft Windows as described in the introduction. According to [20] detection of malware based on PE files can achieve a high detection rate which is exceeding 99 % and a false positive rate below 0.5 %.

In the work [21], the authors used PE files header data to detect malware. Vectors of extracted malware features are then, in one batch with an equal number of benign feature vectors, passed

to the neural network. After processing, n neurons of the output layer form an n-dimensional vector for each input, point in n-dimensional space. On the points obtained is applied k-clustering method to divide benign files from malware into two clusters. Based on the distance between points and cluster centers, the values of neurons are updated. This concludes one round of training. Each step of training uses a new batch of feature vectors mixed with the feature vectors of the previous batch.

After a series of experiments, the authors compared the results of this approach with other recent works based on PE file header features or deep learning. Their method was evaluated as better in most aspects. The better performance against the other PE file header-based malware detection was due to the optimal extraction of features. The headers of PE files were selected based on their effectiveness in differentiating malware from benign files. these assessed as effective were namely NumberOfSections, NumberOfSymbols, or Checksum.

In the article[22] author claims that the detection of malware based solely on selected PE headers and their features can lead to a 99.5 % detection rate, 0.16 % false positive rate, and time below 20 minutes to process data set consisting of 5598 malware samples and 1237 benign files. Another advantage of this approach is that it can detect 0-day vulnerabilities as it is not dependent on an existing AV database according to [20].

Capabilities of SOM clustering can also be used in forensic analysis to separate files of no importance, so-called noise files, from files that can benefit the analysis. In work [23] file metadata is used to train SOM and then compare this method against standard forensic analysis tools. In the work, the authors concluded that the SOM approach is significantly faster because it does not require human eyes to check the patterns and the trained neural network takes little time to produce the output, in this case, data clustered to their respective categories.

Chapter 2
Methodology

2.1 Data collection

For testing, the EMBER dataset [19] is used. The EMBER 2018 dataset consists of 1 million samples collected in or before the year 2018. Each sample has 2381 features extracted from a binary file. The dataset is divided as follows: 200 000 unlabeled samples, 400 000 labeled benign samples, and 400 000 labeled malware samples divided into one of more than 3 000 malware families. The dataset is furthermore divided into training and testing sets. While omitting the unlabeled data items, the training set consists of 600 000 samples and the testing set of 200 000 samples. Samples are equally divided between benign and malicious data. Malware families are not evenly distributed between training and testing sets.

For the purpose of testing classification by SOM, only the labeled samples are used, the unlabeled samples are filtered from the dataset. Also, a custom dataset division into three sets is used to distribute the malware families evenly.

2.2 Data analysis

The four most common, labeled malware families from the Ember dataset are included in the three sets. These are the families: *xtrat* with a total count of 35 969 samples, *zbot* with 24 075 samples, *ramnit* with 20 595 samples, and *sality* with 16 691 samples.

The four family classes together with labeled benign samples form the base of the dataset used in this paper, counting 497 330 samples. However, only a subset of the data is used, as the undersampling technique *RandomUnderSampler* from [24] is used to achieve balanced training and testing dataset.

From the base set, three subsets are created to be used in different parts of the training. The division into three groups is done before any other operation, to prevent the training subset from affecting the testing subset and vice versa. The base dataset is divided into training and testing subsets, both consisting of 50 % of the samples. The division is done on the level of individual classes so that the ratio between benign and malware samples is preserved. Then the training subset is divided again in the same manner to create a validation set, consisting of 50 % samples of the training subset. The training subset is therefore reduced to 50 % of its size.

After dividing the data into three subsets, each subset is normalized using the *StandardScaler* function from [24].

2.3 Dataset limitations

One of the EMBER 2018 dataset limitations, as used in this work, is the data imbalance. The classes of malware chosen as the most numerous ones are significantly smaller in number than the benign class. This can lead to a model that performs well only when detecting benign samples. To remedy this, several approaches are possible. The approach already discussed in the 2.1 section is to reduce the number of benign samples using some undersampling technique. Another approach is to use a classification metric that considers the data imbalance, such as F1-score.

The dataset consists of malware and benign samples recognized and analyzed before 2019. The model is trained and evaluated using such data that it may not perform well on newer malware classes. However, this work does not aim to create a general detection tool but to show the possibilities of using SOM to classify malware.

2.4 The training process

The training process consists of several phases. First, the data vectors are selected from the dataset and divided into three subsets as described in 2.1. Then normalization is applied to scale the data items. PCA is then used to reduce the number of components. To even the number of data items in each class, the undersampling technique is used. From the three subsets, the training set is selected and used as the input to the SOM. This process is displayed in figure 2.1.

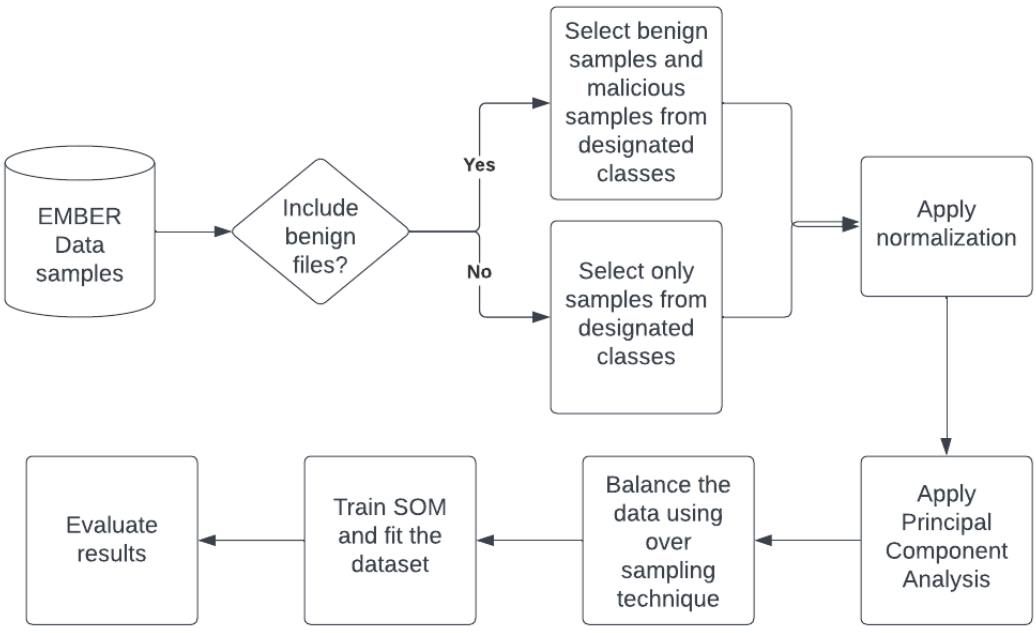

Figure 2.1 Training process of the SOM

2.5 Implementation

For the purposes of this work, the SOM was implemented using C++ programming language. The main functionality of the map is the training and then the evaluation of input items. As the input, the program accepts *Numpy* array and several parameters, that defines the map and the training phase. All of the parameters accepted by the SOM are:

`input_data` (*Numpy* array): the training set.

`map_size` (unsigned int): the size of one side of the map, the resulting number of nodes depends on the map size and on the shape of the map.

`epochs` (unsigned int): the number of training epochs, each one consisting of a number of iterations.

`learning_rate` (double): defines the amount by which the weights are updated, and decrease over time.

`radius` (double): determines the size of the neighborhood of the BMU that is updated. As well as the learning rate, decreases over time.

`shape` (string): the shape of the map can be either square or hexagonal. Together with map size defines the number of nodes.

`distance_metric` (string): metric used to measure the distance between nodes. Supported distances are Euclidean, Manhattan, and Chebyshev.

`nodes_initialization` (string): defines how the nodes of the map are initially set. Supported methods are random and sample.

`batch_size` (int): defines the number of data items processed at once.

`update_learning_rate_function` (string): selects a function that updates the learning rate before each epoch. Supported functions are *linear*, *inverse*, and *decay*.

`calculate_neighborhood_function` (string): selects a function that calculates the neighborhood size for the winning node. Supported functions are *gaussian*, *bubble*.

`update_radius_size_function` (string): selects a function that calculates the radius size before each epoch. The supported function is *decay*.

The learning rate is updated during the training phase before each epoch. The learning rate function should have some properties so that the training process of the SOM is performed in the expected way. The properties are:

- Monotonic decrease - the learning rate decreases over time and the SOM converges to a stable state. Furthermore, the weight updates should be smaller as the training enters the Fine Tuning phase as described in .
- Positive values - the function has positive values so that the weight updates are not inverted. The zero value of the learning rate has no effect on the neuron weights, therefore it is not necessary.
- Smoothness - containing no discontinuities that would lead to unpredictable behavior.
- Convergence to zero - with increasing input values, the learning rate should converge to zero to restrain the training process from further updates that would disrupt the stable state of the SOM.

There are various functions fulfilling these properties. According to [25] the most used are:

`Linear` function is defined as:

$$\alpha(t) = \alpha(0) * \frac{1}{t} \tag{2.1}$$

where t "is the order number of a current iteration"

`Inverse_of_Time` function is defined as:

$$\alpha(t, T) = \alpha(0) * (1 - \frac{t}{T}) \tag{2.2}$$

where T is the number of epochs and t has the same meaning as in 2.1.

`Exponential_decay` function is defined as:

$$\alpha(t, T) = \alpha(0) * e^{\frac{-t}{T}} \tag{2.3}$$

where t and T have the same meaning as in 2.1 and 2.2

Functions 2.1, 2.2, and 2.3 are the supported functions in this implementation. The shape of these functions can be seen in figure 2.2. The initial parameters were set to provide better visualization and do not correspond with the initial settings of SOM parameters. The setting of these parameters is described in one of the next sections.

In the figure is apparent the difference between the three functions. The *Linear* function is quick to decrease and then is slowly converging to zero. This behavior imitates the division of the learning process, giving bigger values for the first phase and smaller values for the second phase. However, the only parameter to influence the progress of the decrease is the initial value of the learning rate, which does not affect the shape of the function.

The *Inverse of Time* function stably decreases over time, fulfilling the conditions of the learning rate function. The function depends on the total number of iterations. However, a smaller number of iterations leads to the learning rate becoming zero earlier than the *Linear* function, while a bigger number of iterations means that the first phase is significantly longer, making greater changes to the topology.

The *Exponential decay* function also depends on the total number of iterations and the decrease of the function can be manipulated by this parameter, as well as by the initial learning rate value. Overall, the decrease is slower than with the *Linear* function and the convergence to zero is slower than the *Inverse of Time* function.

The neighborhood function defines the size of the neighborhood that is updated around the BMU. As well as the learning rate function, it should have some properties for the training phase to perform properly. These properties are:

- Non-negativity - the function gives only positive values or zero as its output so that the neighborhood update is a well-defined value around the BMU or there is no neighborhood and only the BMU is updated.
- Monotonic decrease - as the learning rate function, the neighborhood function should decrease with increasing distance from the BMU as the neurons on the edge of the neighborhood radius are the least influenced by the update.
- Continuous - to avoid big changes in the neurons that are close to each other.
- Symmetry - the update to neurons with the same relative distance to the BMU should be the same to achieve a balanced and stable configuration.

According to [25] the *Bubble* and *Gaussian* functions are widely used in the training process of the SOM.

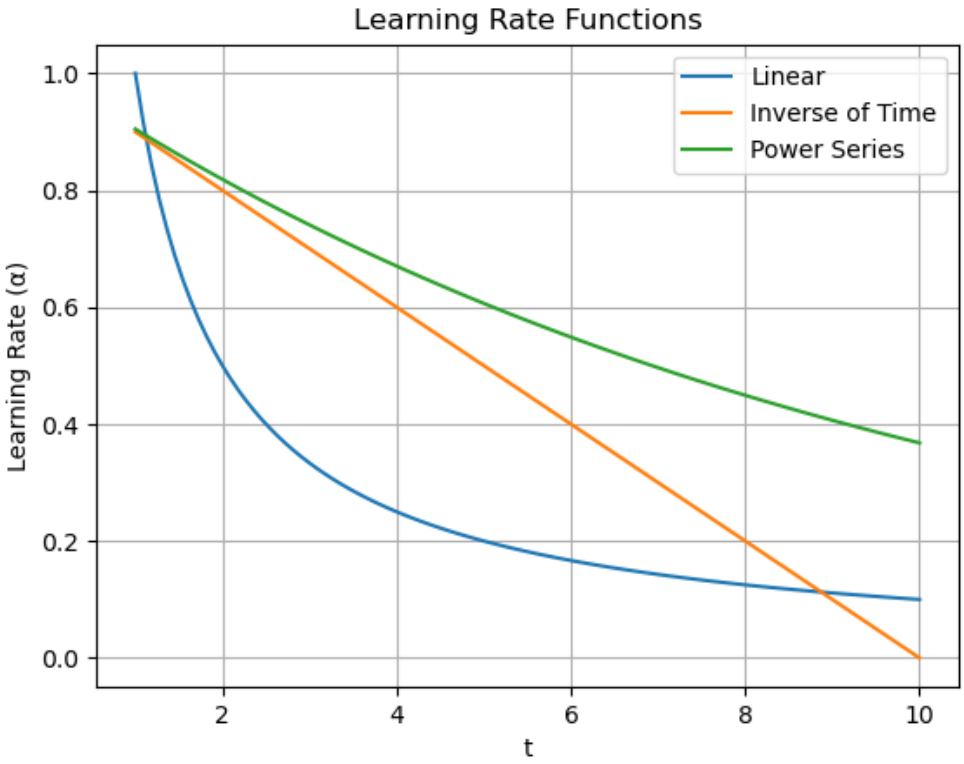

Figure 2.2 Learning rate functions

`Bubble` function is defined as:

$$h_{ij}^{c} = \begin{cases} \alpha(t), (i,j) \in N_c \\ 0, (i,j) \notin N_c \end{cases}$$

where $\alpha(t)$ is the learning rate function and N_c is the set of nodes that are neighboring the node c, which is the node for which the neighborhood function is calculated, the BMU.

`Gaussian` function is defined as:

$$h_{ij}^{c} = \alpha(t) \times e^{\frac{-\|R_c - R_{ij}\|^2}{2 \times (\eta_{ij}^{c}(t))^2}} \tag{2.4}$$

where the function η_{ij}^{c} defines the radius of the neighborhood. The authors of [25] suggest using the exponential decay function described below.

The implemented radius functions that are used in the neighborhood update function are:

`Exponential decay` defined in [25] as:

$$\eta_{ij}^{c}(t) = \eta(0) \times e^{-\frac{t}{T}} \tag{2.5}$$

where $\eta(0)$ is the initial value of the neighborhood radius.

`Interpolation function` suggested by [26] is defined as:

$$\eta_{c}(t) = \eta(0) \times e^{-\frac{t}{\lambda}} \tag{2.6}$$

where $\lambda = \frac{T}{log(\eta(0))}$ is time constant.

2.5.1 Training process

After the nodes are initialized, the training phase takes place. The training process implemented consists of the defined number of epochs, each with a fixed number of inner iterations. In each iteration, items from the input space are presented at random to the SOM, and the weights are updated accordingly. The process implemented is the batch process as described in section.

During the training phase, quantization error (QE) and topological or topographic error (TE) are calculated after each epoch. Values of these errors and their visualization are important markers of the quality of the SOM.

QE "is a statistical metric representing the difference between data and results obtained by letting a Self-Organizing neural network learn the data" [27]. It is the "measure of the average distance between the data points and the map nodes to which they are mapped, with smaller values indicating a better fit" and "Kohonen suggested QE as the basic quality measure for evaluating self-organizing maps" [28]. Such a measure indicates how well the trained map approximates the input data, which is one of the most important features of the SOM. The QE is calculated as follows [28]:

$$QE(M) = \frac{1}{n}\sum_{i=1}^{n}\|\phi(x_i) - x_i\| \tag{2.7}$$

where n is the number of data items in the training set, $\phi : D \mapsto M$ is a mapping function from input space D to the SOM M.

TE is a metric used to evaluate how the topological relationships of the input space are preserved in the trained map as the neighborhood properties of the input data items should be represented in the 2D space of the SOM. According to [28], TE

> is a measure of how well the structure of the input space is modeled by the map. Specifically, it evaluates the local discontinuities in the mapping. TE is calculated as follows [28]:
>
> $$TE(M) = \frac{1}{n}\sum_{i=1}^{n} t(x_i) \tag{2.8}$$
>
> where n is the number of data items in the training set and $t(x_i)$ is a function defined for a data item x, BMU of x $\mu(x)$ and second-best-matching unit of x $\mu'(x)$ as:
>
> $$t(x) = \begin{cases} 0 & \text{if } \mu(x) \text{ and } \mu'(x) \text{ are neighbors} \\ 1 & \text{otherwise} \end{cases}$$

The training process implemented follows these steps:

1. The values of the learning rate and radius are calculated
2. The input data items are shuffled and divided into batches according to the number of available threads, each thread computes one batch
3. For each item in the batch, the BMU is found and the update of weights of the BMU and the neighboring neurons is calculated using the learning rate and radius values and the neighborhood function
4. The weight update is stored in the matrix
5. After all batches are processed, the weight updates are merged with the neuron weights
6. The TE and QE are calculated
7. The epoch number is increased
8. If the epoch number exceeds the number of all epochs or the learning rate drops below the threshold, the training process ends, otherwise continue with the first step

2.5.2 Evaluation process

After the training phase is concluded, the SOM is ready for evaluating data. The function *evaluation* accepts as parameters:

`som` (*Python* dictionary): trained map and configuration settings returned from the *train_map* function

`data` (*Numpy* array): the data items to be evaluated, the testing set

`labels` (*Numpy* array): the labels of the data from the testing set

The data items are then evaluated by finding the BMU for each data item and the label of the data item and BMU is compared. To evaluate the testing set, several metrics are implemented:

Accuracy computed as:

$$\frac{\text{Number of correctly classified instances}}{\text{Total number of instances}} \times 100\%$$

Error rate is the complement of the accuracy.

Precision measures the proportion of true positives among all predicted positives, i.e.

$$\frac{\text{True positives}}{\text{True positives + False positives}}$$

Recall or sensitivity is the measure of true positives among all actual positives, i. e.

$$\frac{\text{True positives}}{\text{True positives + False negatives}}$$

F1 score is a combination of precision and recall and it is calculated as:

$$2 \times \frac{\text{Precision x Recall}}{\text{Precision + Recall}}$$

. It provides a balanced measure of the classifier's performance, ranging from 0 to 1, a higher number indicates higher performance. This metric is useful for imbalanced datasets where accuracy does not provide reliable information about the results of the classification.

For the calculation of the metrics above, the confusion matrix is used. The confusion matrix contains the number of true positives, true negatives, false positives, and false negatives for each class of the dataset.

To expose the SOM as a Python module, the *Pybind11* library is used. Exposed are the default constructor of the SOM, the training method *train_map*, the evaluation method *evaluate*, and the *train_and_evaluate* method.

Chapter 3

Results

3.1 Parameter setting

In this section, the setting of the SOM parameters is described. The SOM parameters are set differently for different types of usage and different types of data. For these reasons, it is important to fine-tune the parameters so that the trained network can classify the malware samples to their respective classes based on their PE headers.

3.1.1 PCA

Before setting the SOM parameters, dimensionality reduction must be applied to the data. As per [29], "PCA converts a group of correlated variables to a group of uncorrelated variables", hence the redundant variables are eliminated by the algorithm.

The PCA algorithm as described by [29] has multiple steps:

1. Standardization of the data:

$$x_j^i = \frac{x_j^i - \overline{x}_j}{\sigma_j} \quad \forall j$$

2. Calculation of the covariance matrix:

$$\sum = \frac{1}{m}\sum_i^m (x_i)(x_i)^T, \sum \in R^{n*n}$$

3. Calculation of the eigenvector and eigenvalue of the covariance matrix:

$$u^T \sum = \lambda\mu$$

$$U = \begin{bmatrix} | & | & | \\ u_1 & u_2 \dots & u_n \\ | & | & | \end{bmatrix}, \quad u_i \in R^n$$

4. Top k eigenvectors of the covariance matrix are chosen:

$$x_i{}^{new} = \begin{bmatrix} u_1^T x^i \\ u_2^T x^i \\ ::::: \\ ::::: \\ u_k^T x^i \end{bmatrix} \in R^k$$

Number of features	Quantization Error	Topological error	Accuracy in percentage
10	4.22	0.62	44.97
15	5.48	0.42	64.72
20	6.57	0.37	70.05
25	7.50	0.31	69.10
30	8.38	0.32	69.49
35	9.39	0.29	67.77
40	10.57	0.29	63.58
50	10.92	0.29	70.25

Table 3.1 PCA selection

Training the map on data items with too many dimensions may lead to longer training times and inferior performance. The exact number of components to be further used was selected by testing various options. The results can be seen in table 3.1. The decisive criteria were QE and overall accuracy. The overall accuracy is simpler to evaluate than the F1 score and can be used in this case, because the dataset used is balanced. The time has been omitted here as it took longer for the map to train on data items with more components.

For the following tests, the value of PCA was chosen as 20, because of the achieved accuracy and quantization and topological errors.

3.1.2 The selection of parameters

3.1.2.1 Size of the map

The size of the map should reflect the size of the input space. According to [3] one neuron per 50 input items is enough. Reflecting this, several values of map size were tested based on the items per neurons ratio. However, the results for a higher ratio showed that the map is not big enough, resulting in large QE and TE. For this reason, bigger maps were tested as well. The *size of the map* variable is the number of nodes on one side of the square map, thus the overall number of nodes is the *size of the map* squared.

As can be seen from table 3.2, the best results, the combination of accuracy percentage and SOM characteristics are with the size of the map set to 100.

3.1.2.2 Number of iterations and learning rate

The number of iterations or epochs defines how many times are the data items presented to the network. To set this parameter, several tests were performed with a combination of the number of iterations and learning rate parameters to choose the best option. The results in the table 3.3 suggest that the combination with the best performance is 100 iterations and 0.01 learning rate.

The QE is a critical metric to select the number of epochs and other parameters, i.e. the size of the map. One of the objectives of the training is to minimize the QE. From the evolution of the error can be decided whether there are enough iterations to train the map. As can be seen from figure 3.1, which visualizes the QE development during 200 epochs, the QE for the learning rate set to 0.1 quickly diverges. Thus the learning rate

Size of the map	Quantization Error	Topological error	Accuracy in percentage
25	9606.72	0.91	20.0
28	664.12	0.93	20.0
34	141.66	0.94	38.75
40	91.84	0.87	37.70
49	43.57	0.87	53.85
68	19.05	0.75	45.96
70	9.02	0.59	50.19
80	6.16	0.55	46.50
90	4.29	0.48	48.07
100	3.43	0.49	67.80
110	2.91	0.53	43.53
120	2.14	0.51	50.83

Table 3.2 Results of items per neuron setting

Number of iterations	Learning rate	Quantization Error	Topological error	Accuracy in percentage
200	0.1	17.08	0.88	32.40
200	0.01	5.37	0.54	68.25
200	0.001	9.84	0.96	40.32
500	0.1	29.47	0.96	21.07
500	0.01	6.93	0.75	67.01
500	0.001	7.28	0.61	64.44
1000	0.1	109.36	0.98	19.18
1000	0.01	6.75	0.84	67.54
1000	0.001	6.34	0.49	69.35

Table 3.3 Number of iterations and learning rate combination test results

of 0.1 has been omitted while measuring the QE for 500 and 1000 epochs, as it distorted the figures.

For the lower learning rates, the overall development can be seen in figure 3.2. Training with a learning rate of 0.01 leads to the lowest QE value after roughly 150 epochs. The value is then slowly increasing and the accuracy is decreasing. The learning rate of 0.001 takes more epochs to reach its minimum, approximately 600 epochs, and after 800 starts to slowly increase, the same as for the previous values of the learning rate.

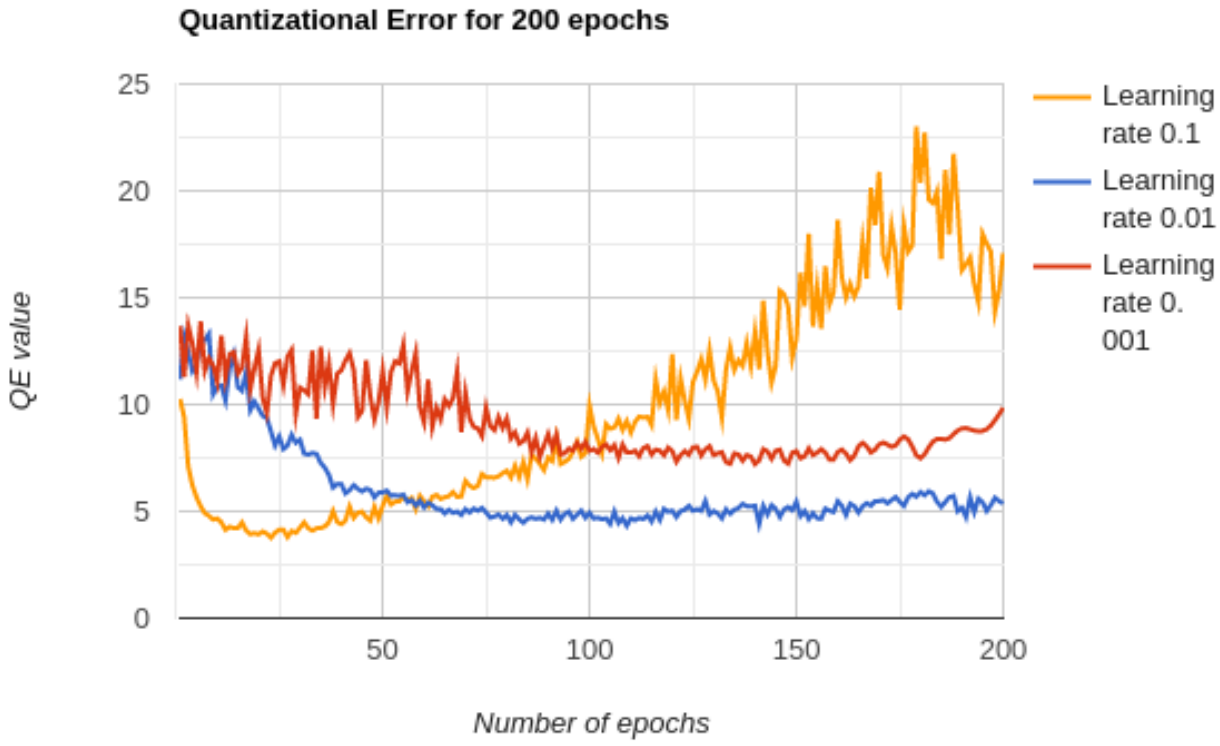

Figure 3.1 Quantization Error for 200 epochs

The TE was evaluated more in detail for the learning rate and the number of iterations as well as the QE. This is because the topographical error represents how well the map projects the topology of the original data. For this topology to be preserved, there must be enough iterations in the first phase of the learning process. However, too many iterations may lead to overfitting.

For the TE, only the figure for 1000 epochs 3.3 is shown as the development is the same and it is clearly visible on the 1000 epochs figure, unlike the figures for the QE. The quick decrease in the TE for the learning rate of 0.1 accounted for the great changes in the topology caused by the bigger adjustment to the weights. This means that a lower learning rate is necessary.

The lower learning rate of 0.01 reached the minimum around 60 epochs, which is again too soon to end the training, as the QE values do not converge to their minimums before 100 epochs. The possible solutions for this issue are either setting the number of iterations to a point where the TE and QE combined are at their lowest, i.e. 200 epochs, or using

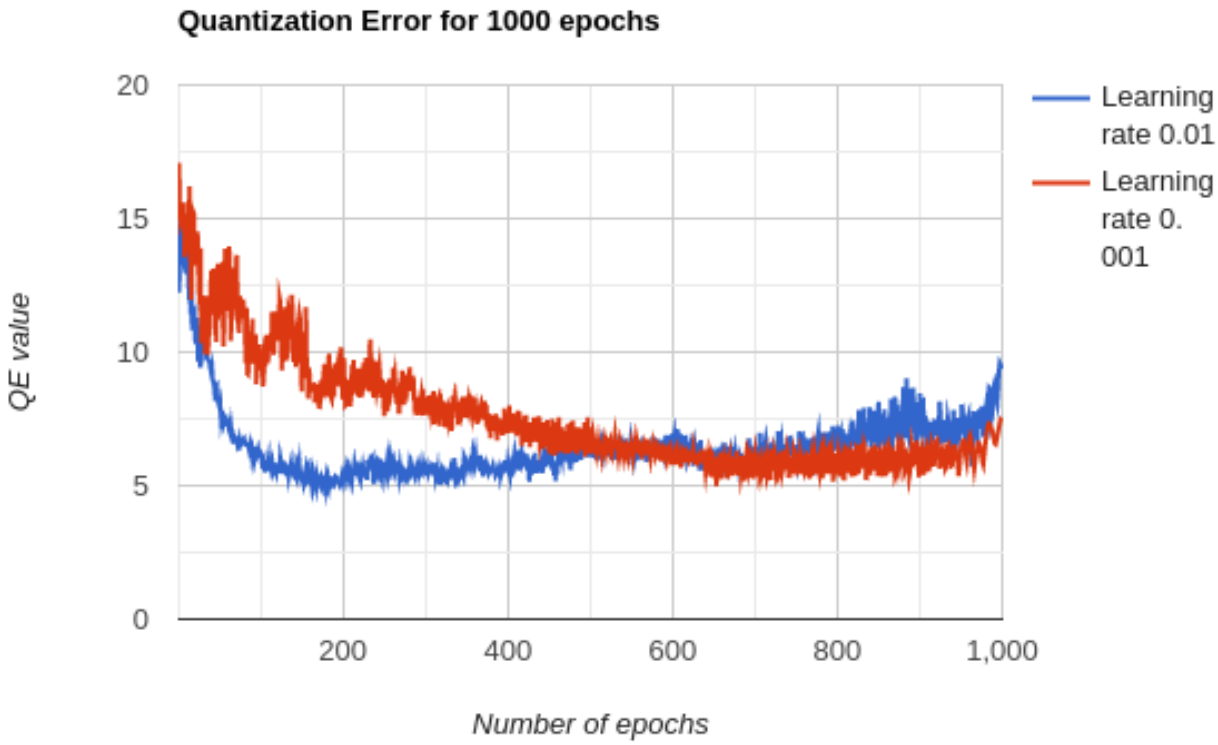

Figure 3.2 Quantization Error for 1000 epochs

a learning rate of 0.001. As for the latter solution, the TE, as well as QE, reached its minimum after 600 epochs. However, the TE is significantly higher for the learning rate of 0.001 after 600 epochs than it is for the learning rate of 0.01 after 60 epochs. The QE is similar for both values.

3.1.2.3 Neighborhood distance

The initial value of neighborhood distance setting affects the first phase of the learning process, as it decreases over time to just a single unit in the end. Bigger values of this parameter mean greater topological changes to the SOM. The important metrics to evaluate when testing the neighborhood distance are the map characteristics, the quantization, and topological errors. The topological error strongly depends on the initial neighborhood distance and is measured in detail again.

Best results in terms of accuracy were achieved for the value of 50 and 20, as can be seen from table 3.4, even though the TE for these values is higher than for the bigger values of neighborhood distance. Figure 3.4 shows that the development of TE for various values of neighborhood distance copies the same pattern that can be seen in figure 3.3, and the values for the neighborhood distance bigger than 10 are similar. However, as was already discussed, the number of epochs should be at least 200 to sufficiently adjust the map to the input values.

Additionally, the neighborhood distance should at the first phase allow the algorithm to

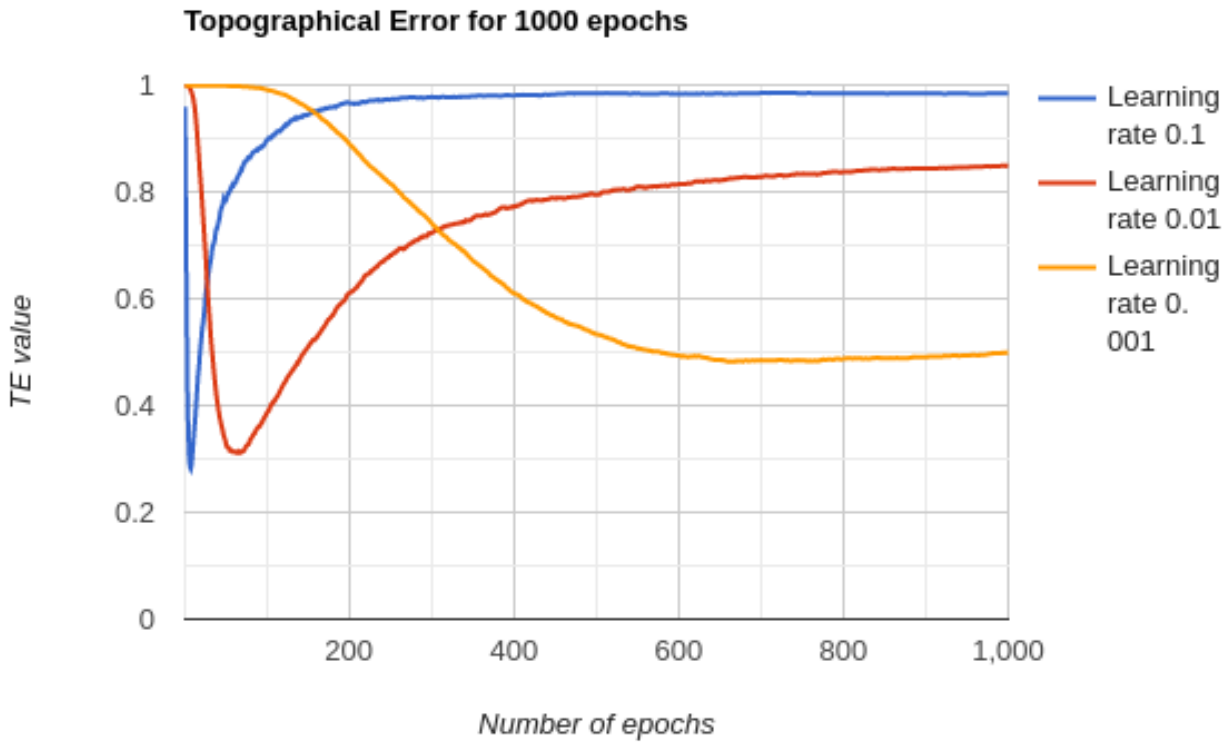

Figure 3.3 Topographical Error for 1000 epochs

make the necessary topological changes, thus it is recommendable to start with such a value that the area includes at least half of the map. This is achieved with the initial neighborhood distance of 50 with the size of the map previously set to 100.

3.1.2.4 Weight initialization

The nodes of the untrained map can be initialized in several ways before the training process. The first option is to give each node a vector of random values from the interval $\langle 0, 1\rangle$ as its initial value. This method is fast, however, an initialization out of the scope of the training sample values can slow or otherwise disrupt the learning process. The normalization of data items should partially prevent this.

The second option is to initiate nodes with random values from the interval $\langle -1, 1\rangle$. This method is better suited to data items containing negative values. Aside from this, the disadvantages are the same as the previous method. This method is referred to as *Random negative*

The third option is to set the initial values of SOM nodes to random sample values from the training set. This option ensures that the initial values are within the range of the training set. Depending on the sample selection, it may be more computationally demanding.

Based on the results of testing 3.5, the method chosen for this purpose is *random negative* weight initialization, although the differences in the topological properties between the

Initial neighborhood	Quantization Error	Topological error	Accuracy in percentage
5	6.97	0.90	64.72
10	5.29	0.79	65.35
20	5.18	0.63	72.25
30	5.35	0.62	53.94
40	5.76	0.59	53.81
50	5.03	0.61	72.79
60	5.94	0.51	69.87
70	6.93	0.51	68.06
80	6.49	0.49	53.80

Table 3.4 Neighborhood distance results

Weight initialization method	Quantization Error	Topological error	Accuracy in percentage
Random positive	5.00	0.57	72.54
Random negative	7.09	0.54	72.95
Sample	7.37	0.55	62.09

Table 3.5 Weight initialization

three methods are small.

3.1.2.5 Distance metric

The tested distance metrics were Euclidean, Manhattan, and Chebyshev, all previously described.

From table 3.6 can be seen, that the best results in terms of accuracy were achieved for the Euclidean and Chebyshev distances. For the final testing, the Chebyshev distance was chosen because of the better topological properties.

3.1.2.6 Learning rate function

As the learning rate function, the following options were tested: *Linear*, *Decay*, and *Inverse*.

From the table 3.7 can be seen that the best option for the data used is the *Inverse* learning rate function.

Distance metric	Quantization Error	Topological error	Accuracy in percentage
Euclidean	5.21	0.47	80.82
Manhattan	20.66	0.47	78.27
Chebyshev	4.09	0.44	80.78

Table 3.6 Distance metric

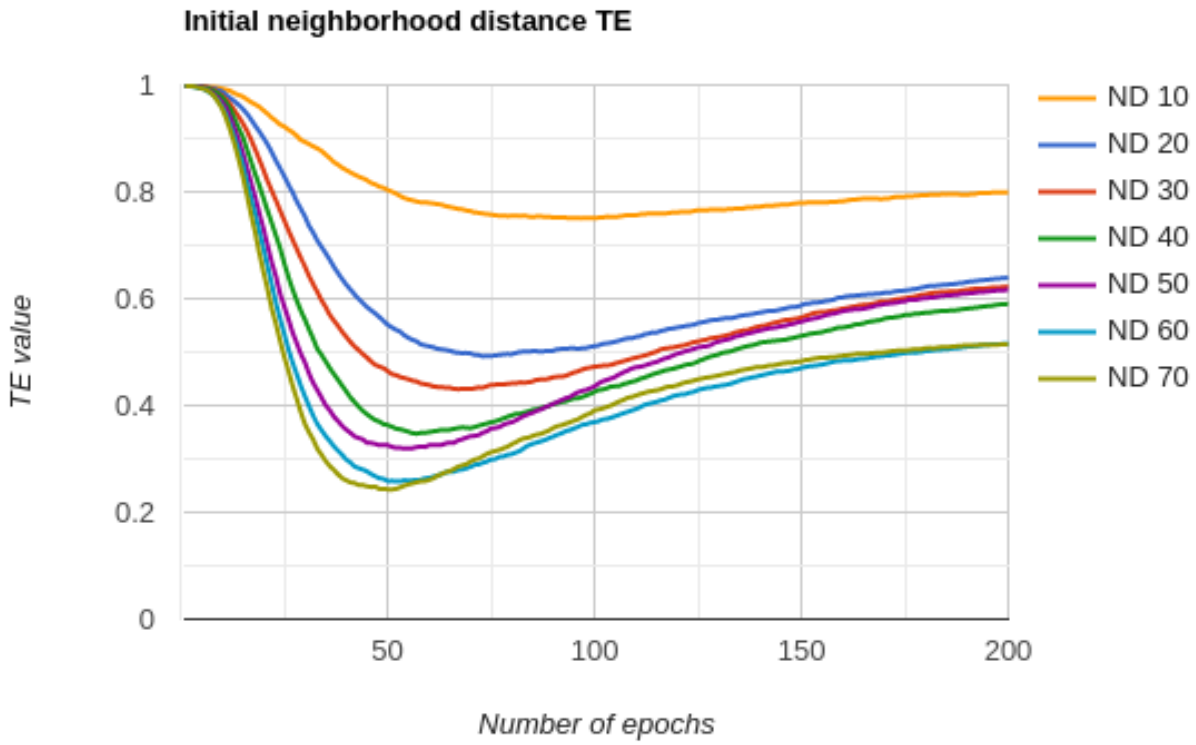

Figure 3.4 Neighborhood distance

3.1.2.7 Neighborhood function

The neighborhood function was chosen amongst the *Gaussian* and *Bubble* functions.

The *Gaussian* function is a better choice according to the results from the table 3.8.

3.1.2.8 Radius function

The radius function used to calculate the neighborhood function was chosen amongst the *Decay* and *Interpolate* functions.

Although the *Decay* function resulted in better topological properties, the overall accuracy was significantly better with the *Interpolate* function. However, as the function is a part of the neighborhood function, the combinations of these functions were tested too.

Learning function	Quantization Error	Topological error	Accuracy in percentage
Linear	7.01	0.82	75.02
Decay	7.43	0.49	80.60
Inverse	4.05	0.55	84.24

Table 3.7 Learning rate function

Neighborhood function	Quantization Error	Topological error	Accuracy in percentage
Gaussian	5.42	0.45	84.42
Bubble	7.43	0.49	82.11

Table 3.8 Neighborhood function

Radius function	Quantization Error	Topological error	Accuracy in percentage
Decay	6.46	0.45	77.36
Interpolate	7.31	0.49	84.84

Table 3.9 Radius function

From the table 3.10 can be seen that the best results were achieved with the neighborhood function *Bubble* and radius function *Interpolate*. For this reason, these functions were used for further evaluation.

3.2 Malware classification results

The SOM was trained with the training dataset and with the parameters that resulted from the testing process. The evaluation was done with the testing dataset. The training and evaluation were performed multiple times and the best results were taken. The results can be seen in table 3.11.

Both types of data were evaluated, malware families only and malware families with benign files. The F1 score shows partial results for each class. The classes in order are: benign, *xtrat*, *zbot*, *ramnit*, and *sality*.

3.2.1 Other ML methods

For comparison, other machine ML methods were used to classify the same data. Also, the same dataset was used to train the models. The methods used were the K-nearest

Neighborhood function	Radius function	Quantization Error	Topological error	Accuracy in percentage
Gaussian	Decay	2.66	0.83	67.17
Gaussian	Interpolate	3.05	0.86	46.32
Bubble	Decay	3.16	0.80	52.61
Bubble	Interpolate	4.11	0.77	72.02

Table 3.10 Neighborhood and Radius functions

Files	Accuracy in percentage	F1 score
Malware and benign	57.68	0.54898011, 0.86265761, 0.5622651, 0.45501348, 0.43317316
Malware only	67.18	0.98234297, 0.81148417, 0.49576172, 0.38267194

Table 3.11 Results comparison

neighbors (KNN) and the Deep Neural Network algorithm, specifically the Multi-layer Perceptron classifier (MLP).

The KNN rule is a classification algorithm used in statistical pattern recognition, according to [30]. The classification is based on a majority vote of k sample vectors that are the closest to the unknown sample. The authors further claim that "the k-NN rule has become the standard comparison method against which any new classifiers, e.g. neural networks, are compared" [30].

The MLP algorithm is a feed-forward artificial neural network providing an output vector for any input vector. "The MLP can be defined as a directed graph with multiple node layers, where the input layer is on the bottom, the output layer is on the top, and the others in the middle are the hidden layers." [31] The network usually contains at least one hidden layer and is, therefore, able to process nonlinear data, contrary to a single-layer perceptron.

For both algorithms, the Scikit-learn libraries were used [24], specifically *KNeighborsClassifier* and *MLPClassifier*.

ML method	Dataset	Accuracy in percentage
KNN	malware families	84.98
MLP	malware families	91.57
KNN	malware and benign files	85.02
MLP	malware and benign files	91.89

Table 3.12 Results comparison

The results of both methods can be seen in table 3.12. For the KNN, odd values of k from 1 to 9 were tried and the best result among them was chosen. The result in the table is for the value of k equal to 3 and the default settings of other parameters. For the MLP, several combinations of sizes of hidden layers were tested. The best result was achieved for the hidden layers of sizes 200 and 300 and the default settings of other parameters.

Chapter 4

Discussion

In this work, the self-organizing map was trained in order to classify malware samples from the EMBER datasets. The parameters of the map were chosen based on a series of tests. For the fine-tuning of the parameters, a training set consisting of the benign samples and the four malware families, the most common in the training set was used. The trained SOM was then evaluated on the EMBER testing dataset. Also, with the same parameters, another SOM was trained on the previous training data without the benign samples, i.e. with the four malware families only.

The SOM is able to cluster the data items into groups by their PE headers. An example of the clustering can be seen in figure 4.1. The size of the map shown, as well as the number of input data items was reduced to improve visibility. In the figure are apparent clusters with irregularities.

For comparison, the k-Nearest Neighbors algorithm and Deep Neural Network were used to classifying the same data. The SOM classifier compared worse than the KNN and the DNN machine learning methods as can be seen from table 3.12. Slightly better results were obtained by classifying malware and benign files together rather than classifying individual malware families.

To improve the results of the SOM, other machine learning methods may be applied to the SOM output, such as KNN or DNN. Those methods could further smooth the irregularities shown in figure 4.1 in the clusters made by the SOM and achieve higher performance. A combination of these methods may lead to a higher detection rate than the KNN or DNN alone may provide, as the SOM preprocess the input of these methods, significantly reducing the dimensionality.

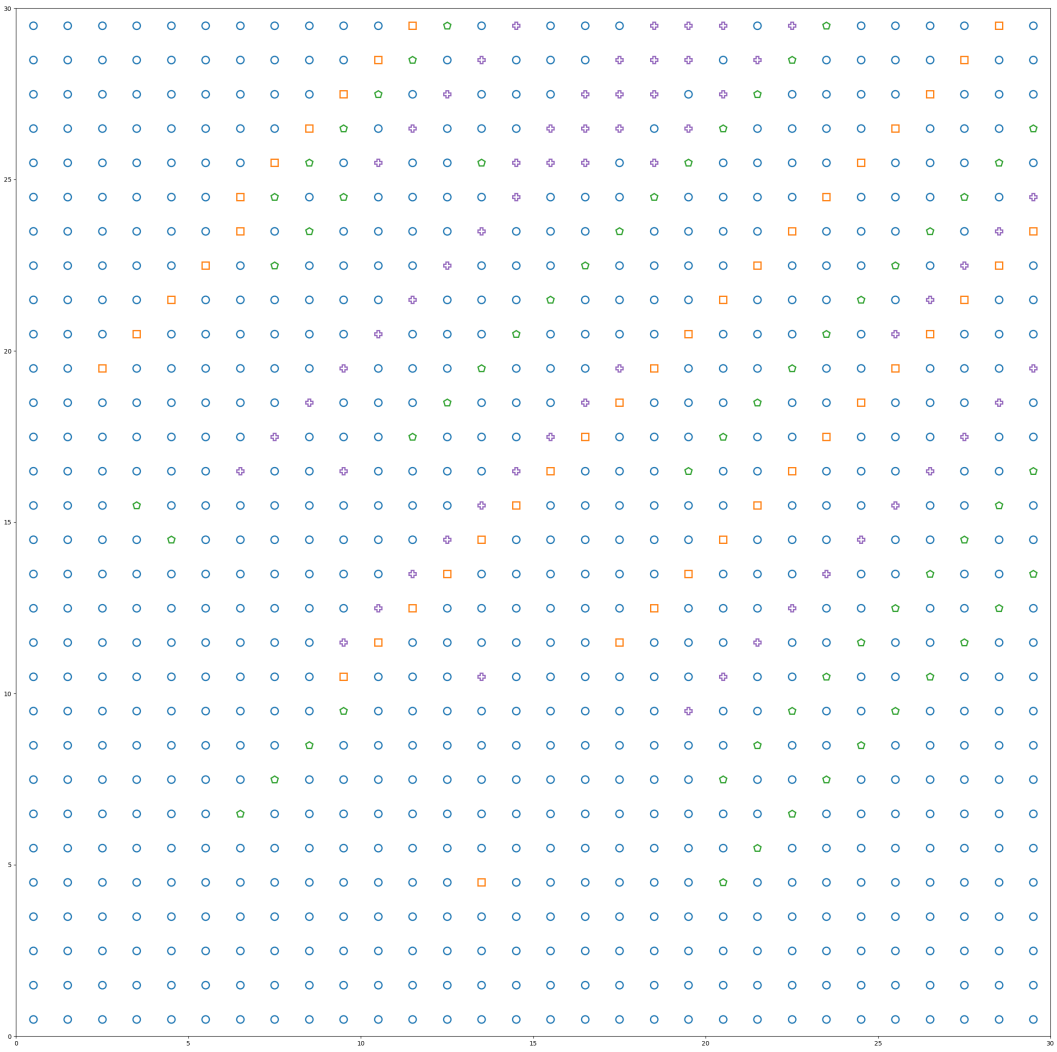

Figure 4.1 Trained SOM

Bibliography

1. OR-MEIR, Ori; COHEN, Aviad; ELOVICI, Yuval; ROKACH, Lior; NISSIM, Nir. Pay Attention: Improving Classification of PE Malware Using Attention Mechanisms Based on System Call Analysis. In: *2021 International Joint Conference on Neural Networks (IJCNN)*. 2021, pp. 1–8. ISSN 2161-4407. Available from DOI: 10.1109/IJCNN52387.2021.9533481.
2. ASLAN, Ömer Aslan; SAMET, Refik. A Comprehensive Review on Malware Detection Approaches. *IEEE Access*. 2020, vol. 8, pp. 6249–6271. ISSN 2169-3536. Available from DOI: 10.1109/ACCESS.2019.2963724.
3. KOHONEN, Teuvo. Self-organized formation of topologically correct feature maps. *Biological Cybernetics*. 1982, vol. 43, no. 1, pp. 59–69. ISSN 0340-1200, ISSN 1432-0770. Available from DOI: 10.1007/BF00337288.
4. MILJKOVIĆ, Dubravko. Brief review of self-organizing maps. In: *2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. 2017, pp. 1061–1066. Available from DOI: 10.23919/MIPRO.2017.7973581.
5. KALTEH, A. M.; HJORTH, P.; BERNDTSSON, R. Review of the self-organizing map (SOM) approach in water resources: Analysis, modelling and application. *Environmental Modelling & Software*. 2008, vol. 23, no. 7, pp. 835–845. ISSN 1364-8152. Available from DOI: 10.1016/j.envsoft.2007.10.001.
6. KOHONEN, Teuvo. Essentials of the self-organizing map. *Neural Networks*. 2013, vol. 37, pp. 52–65. ISSN 0893-6080. Available from DOI: 10.1016/j.neunet.2012.09.018.
7. ERWIN, E.; OBERMAYER, K.; SCHULTEN, K. Self-organizing maps: ordering, convergence properties and energy functions. *Biological Cybernetics*. 1992, vol. 67, no. 1, pp. 47–55. ISSN 1432-0770. Available from DOI: 10.1007/BF00201801.
8. COTTRELL, M.; FORT, J. C.; PAGÈS, G. Theoretical aspects of the SOM algorithm. *Neurocomputing*. 1998, vol. 21, no. 1, pp. 119–138. ISSN 0925-2312. Available from DOI: 10.1016/S0925-2312(98)00034-4.
9. KIANG, Melody Y. Extending the Kohonen self-organizing map networks for clustering analysis. *Computational Statistics & Data Analysis*. 2001, vol. 38, no. 2, pp. 161–180. ISSN 0167-9473. Available from DOI: 10.1016/S0167-9473(01)00040-8.
10. MURTAGH, F.; HERNÁNDEZ-PAJARES, M. The Kohonen self-organizing map method: An assessment. *Journal of Classification*. 1995, vol. 12, no. 2, pp. 165–190. ISSN 1432-1343. Available from DOI: 10.1007/BF03040854.

11. PIRSCOVEANU, Radu-Stefan; STEVANOVIC, Matija; PEDERSEN, Jens Myrup. Clustering analysis of malware behavior using Self Organizing Map. In: *2016 International Conference On Cyber Situational Awareness, Data Analytics And Assessment (CyberSA)*. 2016, pp. 1–6. Available from DOI: 10.1109/CyberSA.2016.7503289.
12. BAILEY, Michael; OBERHEIDE, Jon; ANDERSEN, Jon; MAO, Z. Morley; JAHANIAN, Farnam; NAZARIO, Jose. Automated Classification and Analysis of Internet Malware. In: *Recent Advances in Intrusion Detection*. Ed. by KRUEGEL, Christopher; LIPPMANN, Richard; CLARK, Andrew. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, vol. 4637, pp. 178–197. Lecture Notes in Computer Science. ISBN 978-3-540-74319-4. ISSN 0302-9743, ISSN 1611-3349. Available from DOI: 10.1007/978-3-540-74320-0_10.
13. PIRSCOVEANU, Radu S.; HANSEN, Steven S.; LARSEN, Thor M. T.; STEVANOVIC, Matija; PEDERSEN, Jens Myrup; CZECH, Alexandre. Analysis of Malware behavior: Type classification using machine learning. In: *2015 International Conference on Cyber Situational Awareness, Data Analytics and Assessment (CyberSA)*. 2015, pp. 1–7. Available from DOI: 10.1109/CyberSA.2015.7166115.
14. AV-TEST. *Malware Statistics & Trends Report — AV-TEST* [https://www.av-test.org/en/statistics/malware/]. 2022. Available also from: https://www.av-test.org/en/statistics/malware/.
15. TAVALLAEE, Mahbod; BAGHERI, Ebrahim; LU, Wei; GHORBANI, Ali A. A detailed analysis of the KDD CUP 99 data set. In: *2009 IEEE Symposium on Computational Intelligence for Security and Defense Applications*. 2009, pp. 1–6. Available from DOI: 10.1109/CISDA.2009.5356528.
16. ARP, Daniel; SPREITZENBARTH, Michael; HUEBNER, Malte; GASCON, Hugo; RIECK, Konrad. Drebin: Efficient and Explainable Detection of Android Malware in Your Pocket. 2014.
17. KUMAR, Ajit. *ClaMP (Classification of Malware with PE headers)* [Mendeley Data]. 2020. Version V1. Available from DOI: 10.17632/xvyv59vwvz.1.
18. LASHKARI, Arash Habibi; A.KADIR, Andi Fitriah; GONZALEZ, Hugo; MBAH, Kenneth Fon; A. GHORBANI, Ali. Towards a Network-Based Framework for Android Malware Detection and Characterization. In: *2017 15th Annual Conference on Privacy, Security and Trust (PST)*. 2017, pp. 233–23309. Available from DOI: 10.1109/PST.2017.00035.
19. ANDERSON, H. S.; ROTH, P. EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models. *ArXiv e-prints*. 2018. Available from arXiv: 1804.04637 [cs.CR].
20. SHAFIQ, M. Zubair; TABISH, S. Momina; MIRZA, Fauzan; FAROOQ, Muddassar. Pe-miner: Mining structural information to detect malicious executables in realtime. In: *International workshop on recent advances in intrusion detection*. Springer, 2009, pp. 121–141.
21. REZAEI, Tina; MANAVI, Farnoush; HAMZEH, Ali. A PE header-based method for malware detection using clustering and deep embedding techniques. *Journal of Information Security and Applications*. 2021, vol. 60, p. 102876. ISSN 2214-2126. Available from DOI: 10.1016/j.jisa.2021.102876.
22. LIAO, Yibin. Pe-header-based malware study and detection. *Retrieved from the University of Georgia: http://www.cs.uga.edu/liao/PE_Final_Report.pdf*. 2012.

23. TORASKAR, Tanmay; BHANGALE, Ujwala; PATIL, Suchitra; MORE, Neelkamal. Efficient Computer Forensic Analysis Using Machine Learning Approaches. In: *2019 IEEE Bombay Section Signature Conference (IBSSC)*. 2019, pp. 1–5. Available from DOI: 10.1109/IBSSC47189.2019.8973099.

24. PEDREGOSA, F.; VAROQUAUX, G.; GRAMFORT, A.; MICHEL, V.; THIRION, B.; GRISEL, O.; BLONDEL, M.; PRETTENHOFER, P.; WEISS, R.; DUBOURG, V.; VANDERPLAS, J.; PASSOS, A.; COURNAPEAU, D.; BRUCHER, M.; PERROT, M.; DUCHESNAY, E. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*. 2011, vol. 12, pp. 2825–2830.

25. NATITA, W.; WIBOONSAK, W.; DUSADEE, S. Appropriate Learning Rate and Neighborhood Function of Self-organizing Map (SOM) for Specific Humidity Pattern Classification over Southern Thailand. *International Journal of Modeling and Optimization*. 2016, vol. 6, no. 1, pp. 61–65. ISSN 20103697. Available from DOI: 10.7763/IJMO.2016.V6.504.

26. TU, Le Anh. Improving Feature Map Quality of SOM Based on Adjusting the Neighborhood Function. In: *Sustainability in Urban Planning and Design*. IntechOpen, 2019. ISBN 978-1-83880-352-0. Available from DOI: 10.5772/intechopen.89233.

27. WANDETO, John M; DRESP-LANGLEY, Birgitta. The quantization error in a Self-Organizing Map as a contrast and color specific indicator of single-pixel change in large random patterns. [N.d.].

28. BREARD, Gregory. *Evaluating Self-Organizing Map Quality Measures as Convergence Criteria*. Kingston, RI, 2017. Available from DOI: 10.23860/thesis-breard-gregory-2017. PhD thesis. University of Rhode Island.

29. REDDY, G. Thippa; REDDY, M. Praveen Kumar; LAKSHMANNA, Kuruva; KALURI, Rajesh; RAJPUT, Dharmendra Singh; SRIVASTAVA, Gautam; BAKER, Thar. Analysis of Dimensionality Reduction Techniques on Big Data. *IEEE Access*. 2020, vol. 8, pp. 54776–54788. ISSN 2169-3536. Available from DOI: 10.1109/ACCESS.2020.2980942.

30. LAAKSONEN, J.; OJA, E. Classification with learning k-nearest neighbors. In: *Proceedings of International Conference on Neural Networks (ICNN'96)*. 1996, vol. 3, 1480–1483 vol.3. Available from DOI: 10.1109/ICNN.1996.549118.

31. WAN, Shaohua; LIANG, Yan; ZHANG, Yin; GUIZANI, Mohsen. Deep Multi-Layer Perceptron Classifier for Behavior Analysis to Estimate Parkinson's Disease Severity Using Smartphones. *IEEE Access*. 2018, vol. 6, pp. 36825–36833. ISSN 2169-3536. Available from DOI: 10.1109/ACCESS.2018.2851382.

Contents of the media attached

- `readme.txt` ..... brief description of the media contents
- `exe` ..... directory with a compiled module
- `src`
  - `impl` ..... source code of the implementation
  - `thesis` ..... source code of the work in the LaTeX format
- `text` ..... text of the work
  - `thesis.pdf` ..... text of the work in the PDF format