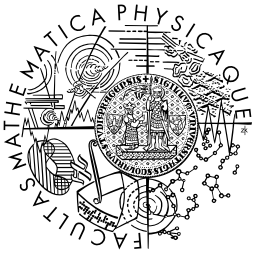

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Jan Bodnár

Application of artificial neural networks for malware detection in HTTPS traffic

Department of Software Engineering

Supervisor of the bachelor thesis:	Jakub Lokoč Ph.D.
Study programme:	Computer Science
Study branch:	General Computer Science

Prague 2017

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

Title: Application of artificial neural networks for malware detection in HTTPS traffic

Author: Jan Bodnár

Department: Department of Software Engineering

Supervisor: Jakub Lokoč Ph.D., Department of Software Engineering

Abstract: A huge proportion of modern malicious software uses Internet connections. Therefore, it is possible to detect infected computers by inspecting network activity. Since attackers hide the content of communication by communicating over encrypted protocols such as HTTPS, communication must be analysed purely on the basis of metadata. Cisco provided us a dataset containing aggregated metadata with additional information as to whether or not each sample contains malicious communication. This work trains neural networks to distinguish between infected and benign samples, comparing different architectures of neural networks and providing a comparison with results achieved by different machine learning methods tried by colleagues. It also seeks to create a mapping which maps samples of communication into a space where different samples of malicious communication created by a single malware family form clusters. This may make it easier to find different computers infected by a virus with known behaviour, even when the virus cannot be detected by the detection system.

Keywords: artificial neural networks, malware detection, HTTPS traffic, similarity search

I would like to express my gratitude to my supervisor Jakub Lokoč Ph.D. for his guidance, to our team for their cooperation, and to the Cisco company for making this thesis possible.

I would also like to thank my family for their continued support.

Contents

1. Introduction

1.1 Motivation

Some decades ago, computer viruses were written by a small number of skilled individuals for fun and fame. As time passed, the affordability and with it the number of computers grew, and consequently the amount of important infrastructure, and data being moved on it, also increased.

This rapid growth of the importance of computers together with the growing percentage of computers connected to the Internet caused an important shift in the attitude of hackers. They realized how powerful their knowledge had become and so they started to use it to make profit. Later, as computer systems and malware became more complex, hackers started to organize. Nowadays, many hackers operate in groups, whose size and organizational structures can easily be compared with smaller software companies.

Those "software companies" are working day and night to hack others, steal their data, blackmail them, damage their business or simply use their computers for various purposes such as sending spam, carrying out DDoS attacks or mining cryptocurrencies.

Most of the mentioned activities require malware to use the Internet connection. This makes it possible to detect infection of a computer by inspecting the network traffic.

1.2 Justification

One possible way to detect infected computers by analysis of their network communication is to directly inspect the communication's content. We can start looking inside packets for groups of strings which are known to be specific to a given malware. Unfortunately, authors of malware easily can, and do, prevent this type of detection by communicating over a protocol employing encryption. Arguably, the protocol most commonly used for this purpose is HTTPS, since its frequent use by benign applications prevents it from being blocked.

It is difficult to detect malicious communication within HTTPS, because we cannot rely on analysis of information obviously present in communication.

Connection to a previously unknown host cannot automatically be considered suspicious, because load balancing frequently occurs on popular websites.

Analysis of used certificates and certificate authorities is also unlikely to provide much help. In the past, it was hard for attackers to get a valid certificate signed by a trusted authority because buying a certificate was a complex procedure involving payment for the certificate and sometimes even telephone screenings and other procedures. Hackers either had to undergo such procedures and try to pay for the certificate anonymously, or they had to steal a certificate from some previously infected computer. This made generation of certificates risky and difficult to automate. Nowadays attackers' position is easier because there exist certification authorities, such as Let's Encrypt, which are trusted by browsers and used to sign regular domains, but which also sign certificates to anybody, without any further controls and for free. Thus authors of malware can

automatically generate certificates signed by the same certification authority as legitimate traffic.

One way to directly inspect HTTPS communication is by utilizing a technique resembling a man-in-the-middle attack. We can install our software between the end-user and the Internet. Each installation of our product will have a separate certification authority, which can be added to the list of trusted authorities on the end-user's computer. Then whenever the user wants to open a TLS/SSL connection to a server, the first part of our security software opens one encrypted connection between this server and itself and another part opens encrypted connection between itself and the user. It shows the user a certificate created by the program's own certificate authority so that the user does not notice anything. Then it forwards the user's requests and the server's replies. By this method the software can monitor the communication. This method is currently in use e.g. by Avast, but we decided not to go this way because we consider it too intrusive, since applications which carry their own certificates and do not use certificate authorities can break down. The method also does not allow users to check the check-sums of certificates, which can be a security issue especially for websites using self-signed certificates. Furthermore, we also consider the approach to be difficult to scale.

To successfully detect malicious communication without having to rely on the analysis of communication content, it is necessary to build the detection system on a completely different basis: analysis of behaviour. Thus, we can analyse with whom is the computer interacting, how much data it is transferring, how often mutual interaction occurs, etc., and base the detection on such information. If a computer for instance starts to send large amounts of data to computers with which it has never communicated before, we can consider such an activity suspicious and block it, as malware may be trying to steal our data. Of course, malware authors can try to evade even behaviour-based detection, but this will require much more work, and authors will be forced to completely change the way their malware works.

To detect and block malware on the basis of its behaviour, we have to create the rules on the basis of which our detection system makes decisions. This would be difficult and time-consuming for humans, especially because of the number of positive samples that must not be blocked and the overall difficulty of handling unstructured data. For this reason, in the context of a large amount of data, we can make use of machine learning methods and let the computer find the rules by itself.

Several machine learning classification algorithms are suitable for this task. Some of the most important are k nearest neighbours, random forests and neural networks. This thesis seeks to apply neural networks; other members of our team have tried the remaining methods.

1.3 Studied problems

This work focuses on two different problems. The first is detection of malicious communication in HTTPS communication on the basis of metadata such as the length of data transmitted from client to server or the duration of communication. We were provided a dataset containing labeled samples of metadata created

by both benign and malicious activity. Our task was to teach neural networks to distinguish between the classes and to provide a comparison of performance of different architectures. We also provide a brief comparison with the results achieved by our colleagues on the same task using different machine learning techniques.

Our second task was to explore the methods of similarity search. Upon finding an infected computer, one of the tasks that should be performed is a mapping of the infection to find whether there are other computers which may be infected by the same virus. We try to create a method which helps with this task by trying to group together samples of communication created by the same malware family. This will allow the administrator to take samples of communication from an infected computer and look for other computers with similar network behaviour, thus obtaining a list of computers she should inspect.

The data for this thesis were provided, aggregated and labelled by the Cisco company and originate from measurements in 500 large companies. The network activity was represented as a vector of numbers of a static length. We test three different representations of the traffic developed by Kohout and Pevny [2015] and Tomáš Komárek Kohout et al. [2017].

The questions that are in focus here are whether our representations contain enough information and whether a given machine learning method can make use of the information and learn to distinguish between malicious communication and normal communication. We are not interested in the question of how easily our detection system can be evaded by an adapting adversary; we can therefore make a strong assumption that our adversary will not change her behaviour in time. The question of an adapting adversary has been studied for instance in Samusevich [2016].

2. Data

The data are based on measurements of HTTPS traffic in 500 large companies during two weeks in March and April 2016. The dataset was provided by a team from Cisco, with whom we cooperate in the analysis. In the following chapter we describe the data in greater detail.

2.1 Data content

The goal here is to detect malware which uses HTTPS to mask its communication. Since HTTPS is a protocol employing encryption, it highly limits the amount of useful information we are able to extract from the ongoing communication. We further limit our options by not being willing to collect any sensitive information, such as HTTPS certificates or domain names, while protecting our customers with the trained system – although during the training process we had to collect target IP addresses to make the training possible. Because of this limitation we have to rely purely on collection of metadata.

For each HTTPS connection Cisco collected date, time, source and target IP address, and a quadruple of metadata describing the communication: number of bytes transferred from client to server (m_{up}), number of bytes transferred from server to client (m_{down}), time duration of the connection (m_{dur}) and time interval from the last opening of connection between client and server (m_{in}).

Additionally, they checked whether the host IP address is present in databases of servers spreading malware, and provided each connection with the label Infected or Clean. Unfortunately, this approach does not guarantee that our labels are 100% accurate. We may have missed an infected connection because the host server is not in any of the used databases. Also, since the control servers can be normal servers hacked by the authors of the malware, they may be serving some benign purpose; therefore, not all the communication with them necessarily has to be malicious.

When we look at the data we can see three main problems.

The first is that the amount of information contained in the available data is very limited. Despite this, it is sufficient to successfully detect malicious communication with a very high accuracy, as we will show in Chapter 4.

The second problem is that malicious communication is extremely rare and therefore we have a very limited number of infected samples in our data. This is a big problem and will limit us in the training phase of all the experiments.

The third problem is that the data can hardly be used in this form. We have to use some form of aggregation, which will allow us to compare and analyse larger numbers of requests at the same time.

2.2 Data Aggregation

The researchers from Cisco developed two forms of aggregation. Since the goal of this project is to develop means of detecting infected clients, they aggregated the quadruples on the basis of the client's IP address and the time. An alternative

would be to aggregate the traffic on the basis of the host IP and try to detect infected hosts, but that is not covered in this project. The Cisco researchers created a separate set of quadruples for each five minutes of activity of each client. Each set contained client ID, the label Infected or Clean, and the quadruples $(m_{up}, m_{down}, m_{dur}, m_{in})$, as described above. The resulting data comprise 43,813,135 aggregated sets of clean samples of communication and 2,800 sets of infected communication in the first week, and 31,607,364 clean and 1,578 infected aggregated samples in the second week. These figures show the aforementioned disproportion between the number of clean and infected samples.

We now have to transform the sets into a form which can be handled by machine learning algorithms. Such a representation should have fixed length, should preferably be dense and should preserve information, and the structure of output should resemble the structure of the initial data.

2.3 Soft Histograms

The first such form of aggregation is the "soft" histogram Kohout and Pevny [2015]. The general idea behind this approach is to create one histogram from each set generated above. Each set contains quadruples of form $(m_{up}, m_{down}, m_{dur}, m_{in})$. We first re-scale those quadruples by application of mapping $lg : R^4 \rightarrow < 0, +\infty >^4$ defined as:
$lg((m_{up}, m_{down}, m_{dur}, m_{in})) = (log(1+m_{up}), log(1+m_{down}), log(1+m_{dur}), log(1+m_{in}))$
Since the biggest value in each dimension of the histogram is smaller than 10, it is created as a four-dimensional array $\{0, 1, .., 10\}^4$ with $11^4 = 14641$ bins. Then we create mapping $D()$ from space $\{< 0, 10)^4\}^*$ occupied by sets quadruples into space $\{0, 1, .., 10\}^4$ of the histogram. This type of mapping is usually defined as $D : (< 0, 10 >^4)^* \rightarrow \{0, 1, .., 10\}^4$:
$Histogram[*][*][*][*] = 0;$
For each quadruple x:
Histogram$[\lfloor lg(x) \rfloor]$+=1

The result of this approach is a so-called "hard" histogram. Hard histograms can behave well in a stable environment but our data contain a large amount of noise, such as unpredictable network latency. If a quadruple is projected by the mapping lg to value [2.9, 1.8, 4.49, 1.3], it then modifies the histogram by increasing the value of bin [3,2,4,1]. We can see that even a minor, randomly caused change in the duration of the communication can result in a vector [2.9, 1.8, 4.51, 1.3] being created instead. This results in our algorithm modifying the value of bin [3,2,5,1]. Such behaviour is very problematic, since machine learning algorithms do not usually take into account that bin [3,2,4,1] is close to bin [3,2,5,1]. Therefore, moving the request into a different but nearby bin may result in a completely different output of the machine learning algorithm, which may cause this type of histogram to behave poorly.

This problem can be addressed by making the histogram soft. We can put the information about a request not only into a single bin but also into its neighbours. If we insert a request mapped via lg to a vector u=[2.9, 1.8, 4.3, 1.51] into the histogram, we modify the values of bins in a multidimensional rectangle whose

edges are parallel with axes and whose two vertices are $\lfloor u \rfloor$ and $\lceil u \rceil$. In our example this means bins [w,x,y,z] such that w $\in \{2,3\}$, x $\in \{1,2\}$, y $\in \{4,5\}$ and z $\in \{1,2\}$. We construct the histogram in the following way.

At first, the histogram is filled with zeros. Then, whenever we add a request represented by a quadruple u element of $< 0, 10 >^4$, the following procedure is followed:
1) "Lower" indices $l_k := \lfloor u_k \rfloor$ are computed
2) Contributions of u to "lower" bins
$c_k := 1 - (u_k - l_k)$ are computed.
3) All bins with coordinates
$\{(l_1 + k_1, l_2 + k_2, l_3 + k_3, l_4 + k_4) | (k_1, k_2, k_3, k_4) \in \{0, 1\}^4\}$
are increased by values
$\prod_{j=1}^{4} (c_j^{(1-k_j)} * (1 - c_j)^{k_j})$.

2.4 Gaussian mixture models

The soft histogram representation has two disadvantages: it is very sparse and has very large dimensions. To address these problems, Tomáš Komárek Kohout et al. [2017] applied a Gaussian mixture model technique as a different method of data aggregation.

Gaussian mixture models (GMM) rely on modelling the a priori probability of quadruples m representing each request. They model this probability under the assumption that those quadruples are taken from one of d random processes, each of which produces output with four-dimensional Gaussian distribution with mean value μ and covariance matrix Σ_i. Let G_k denote an event whereby message m is created by the k-th of the processes, and G denote Gaussian distribution. Then, we model the probability of a message m, $P(m)$, as follows:

$P(m) = \sum_{k=1}^{d} P(G_k) * P(m|G_k)$

$P(m) = \sum_{k=1}^{d} P(G_k) * G(m|\mu_k, \Sigma_k)$

Dimensionality reduction via GMM has two phases: in the first phase we train our model to fit our training data; the second phase is inference. We use the trained model to aggregate sets of quadruples into vectors of fixed length. During the training phase, we first select the number of processes d, which will be the number of dimensions of the output vectors. Then we train the model via the expectation maximization algorithm.

This algorithm maximizes the probability P that the random process we describe will generate exactly the quadruples we have seen:
$P = \prod_{i=1}^{number\_of\_quadruples} P(m_i)$
$P = \prod_{i=1}^{number\_of\_quadruples} (\sum_{k=1}^{d} P(G_k) * G(m|\mu_k, \Sigma_k))$
by modification of $P(G_k)$, $\mu_{1..d}$ and $\Sigma_{1..d}$.

During the inference phase we take the sets of quadruples which we are supposed to aggregate and for each of those sets create a d dimensional vector x. Let $\{m_1, ...m_N\}$ denote one of those sets. Then each dimension k of vector x contains average of probabilities that given quadruples were created by the k-th Gaussian process:

$x_k = \frac{1}{N} * \sum_{m=1}^{N} P(G_k|m_i)$

where $P(G_k|m_i)$ is computed as follows:
$P(G_k|m_i) = P(G_k \& m_i)/P(m_i)$

$P(G_k|m_i) = P(m_i|G_k) * P(G_k) / \sum_{j=1}^{d} P(m_i|G_j) * P(G_j)$

During the inference phase we ignore the original probabilities $P(G_k)$ and set them as equal, because $P(G_k)$ is simply a proportion of data created by a given process k and this number likely changes over time.

$P(G_k|m_i) = P(m_i|G_k) / \sum_{j=1}^{d} P(m_i|G_j))$

$P(G_k|m_i) = G(m_i|\mu_k, \Sigma_k) / \sum_{j=1}^{d} G(m_i|\mu_j, \Sigma_j)$

Since this problem is very imbalanced, it is better to train separate Gaussian mixture models for clean data and malicious data. The first part of the $d_c + d_m$-dimensional output vector then contains the averages of probabilities that each quadruple was generated by each of d_c Gaussian distributions modelling the generation of clean communication, and the second part contains the averages of probabilities that each quadruple was generated by one of d_m Gaussian processes producing malicious communication. We simply train one model on malicious communication and one model on clean communication, and in the inference phase we take outputs for each of those models and concatenate them. With this approach we can choose d_c and d_m differently. The data contain two versions of data reduced via GMM: GMM600, which contains 300 elements trained on clean quadruples and 300 elements trained on infected quadruples, and GMM600_480, which contains 480 elements trained on clean quadruples and 120 trained on infected quadruples.

3. Neural networks

Neural networks are a group of machine learning techniques and algorithms. They have a fixed architecture built from small units called neurons, which have trainable numerical parameters. These parameters are trained iteratively. In each training iteration the network receives input data and computes an output value. We then modify the network on the basis of its intermediate results and the difference between its output value and the desired output value, in such a way that if the modified network is run on this data again, its output will be closer to the desired output. This process is supposed to teach the network to accomplish the task using the training dataset. The hope is that if a neural network learns to produce a correct output on the training set, then it will also generalize, i.e. produce correct output for inputs it has never seen before. In the following chapter we describe neural networks in greater detail.

3.1 The perceptron

As we have mentioned, neural networks have a fixed architecture built from smaller trainable components called neurons. A commonly used neuron architecture is the perceptron. We can formally define the perceptron with vector of input data $X \in R^n$, weights $W \in R^n$, bias $b \in R$, and activation function $\sigma : R \to R$ as a mapping $f : R^n \to R^m, f(X) = \sigma(w^T x + b)$. Weights and bias are modified during training. Commonly used activation functions include Sigmoid, TanH (3.1) and recently also ReLU (3.2):
$Sigmoid(x) = \frac{1}{1+e^{-x}}$
$TanH(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
$ReLU(x) = max(0, x)$
Rosasco et al. [2004]

The function of the perceptron can be interpreted as a measurement of distance of input point X from hyperplane $W^T y + b = 0$. The training process then involves modification of the position and orientation of this hyperplane. From this interpretation we can see that the ability of a single perceptron to fit the input data is very limited.

To increase the flexibility of the network we have to use multiple perceptrons at the same time. Most often, the perceptrons are organized in layers in such

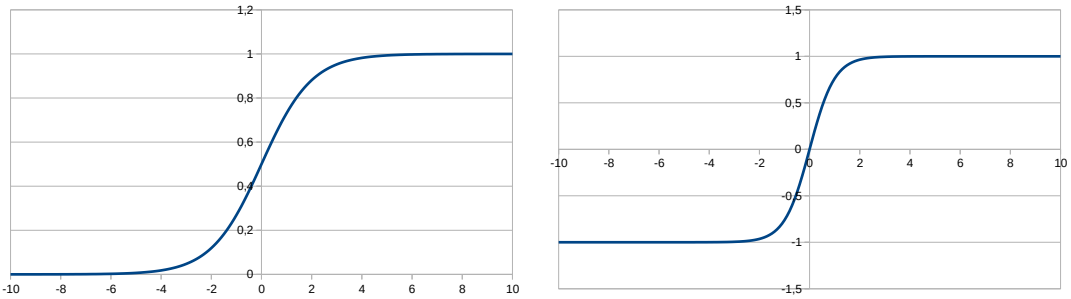

Figure 3.1: Sigmoid (left) and TanH (right) activation functions.

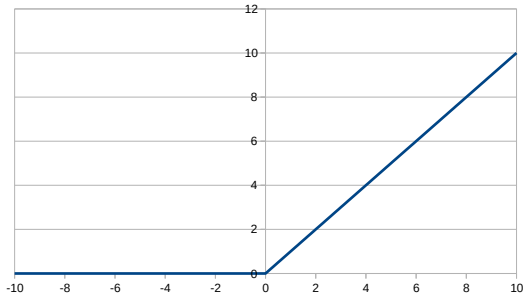

Figure 3.2: ReLU activation function.

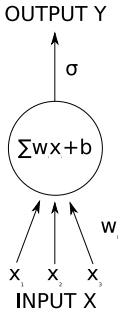

Figure 3.3: Perceptron computes function $\sigma(\sum_{i=1}^{n} w_i * x_i + b)$.

a way that each perceptron in the first layer is connected to each component of the input data, and in every layer each perceptron is connected to all the outputs of the previous layer. The output of the last layer is considered the output of the network. This architecture is called multilayer perceptron. Other ways of organizing neurons include convolution and residual connections, used e.g. in image classification, or recurrence, used for instance in natural language processing or different fields involving operation with series of arbitrary length.

3.2 Training

As mentioned, a neural network is trained via an iterative process. First, we initialize the weights and biases randomly, but in a carefully chosen way, because

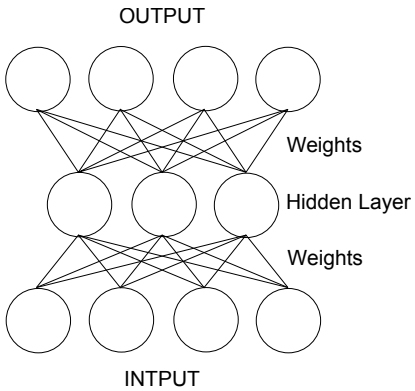

Figure 3.4: A multilayer perceptron consisting of three fully connected layers - input layer, hidden layer and output layer.

this can affect the speed of convergence Kumar [2017]. After the initialization, an iterative process begins. This process is based on so-called "error backpropagation". In each step we select a subset of training data, called a batch; then, we evaluate the network using the data, analysing the network's prediction error, and modify the network in such a way that if we evaluated the network on the same set of data again, its outputs would be closer to the desired outputs and its error would be smaller.

How do we know which parameters to update, and how they should be updated to make the error smaller? First, we need a function calculating a real number which tells us the size of the error in the network's predictions. We will then try to find network parameters which minimize the output of this function. In machine learning such a function is called a "loss function".

One commonly used loss function is called Mean Square Error (MSE) Rosasco et al. [2004]. Let f denote a mapping $R^n \to R^m$ computed by the neural network, $x \in R^n$ input vector and $y^* \in R^m$, the desired output value for input x. The MSE can then be defined as follows:
$MSE(x, y^*, f) = \frac{1}{m} \sum_{i=1}^{m} (f(x)_i - y_i^*)^2)$.
Another common loss function is hinge loss Rosasco et al. [2004], usually used to train two-class classifier networks with $m = 1$ and $y^* \in \{-1, 1\}$:
$HingeLoss(x, y^*, f) = max(0, 1 - f(x) * y^*)$.
In each step we usually train the network on multiple inputs at the same time, and therefore it is useful to define the MSE also for batches. Let $X = (x_1, x_2, x_3, ..., x_k)$ denote the input vectors and $Y^* = (y_1^*, y_2^*, y_3^*, ..., y_k^*)$ desired outputs. Then, the MSE is defined as the average of errors on the inputs from the batch:
$MSE(X, Y^*, f) = \frac{1}{k} \sum_{j=1}^{k} MSE(X_j^*, Y_j^*, f))$
$MSE(X, Y^*, f) = \frac{1}{k} \sum_{j=1}^{k} \frac{1}{m} \sum_{i=1}^{m} (f(X_j)_i - Y_{j,i}^*)^2))$
The hinge loss for batches is defined in exactly the same way.

When we have chosen an error function we can use methods of mathematical analysis. If a neural network and its loss function are composed purely of differentiable parts, we can partially differentiate the whole loss function with respect to all the trainable parameters. In the process of differentiation we take input vectors X as constants. The goal is to compute the gradient (vector of first partial derivatives), showing the direction of the fastest growth of the loss functions with respect to the trainable parameters. If we find a gradient in point specified by input vectors X, desired outputs Y^* and parameters of the neural network, we can multiply the gradient by minus one and thus get the direction of the fastest decrease of the loss function in the current state of the network with regard to the trainable parameters. We can then shift the parameters by a small step in the direction of the minus gradient. If the step is small enough, evaluation of the modified network on the same inputs will show a decrease in error. Formally we can write the change of the i-th neuron' bias B_i and it's j-th weight $W_{i,j}$ as follows:
$W_{i,j} = W_{i,j} - \alpha * \frac{\partial f}{\partial W_{i,j}}(W, B, X)$
$B_i = B_i - \alpha * \frac{\partial f}{\partial B_i}(W, B, X)$

A positive real number α is the size of the modification in the value of the variable in the given direction. It is called the learning rate and its value usually changes during the training. There exist algorithms which change it adaptively

and algorithms which change it in a fixed way (e.g. multiplying alpha by $\frac{1}{2}$ every K iterations).

The algorithm we have just described is "stochastic gradient descent" (SGD) Goodfellow et al. [2016]. Gradient descent means that we descend in the direction of the biggest local decrease of the loss function and stochastic means, trying to find a global optimum of the loss function on the whole training set via modifications of the neural network only on the basis of smaller subsets. Unfortunately, gradient descent-based methods do not guarantee convergence towards a global optimum and usually lead to convergence towards some local optimum instead.

A more advanced algorithm offering faster convergence rate is "Adam" Kingma and Ba [2014]. This algorithm works in the same way as SGD except that it provides momentum which improves its behaviour in unstable environments. It remembers past gradients and combines them with the current gradient. For example, if in the previous step we decreased the weight somewhat and now want to increase it, Adam will increase the variable only by a small proportion.

Although the previous two algorithms converge to a local optimum of loss function on the training dataset, we do not want to reach a state of complete convergence, but only to get close to it. This is because a network trained to this state would be too adapted to the training set and thus would generalize poorly – it would perform badly with data it had never seen before. This effect is called overfitting and is a common problem when training neural networks.

Overfitting becomes a bigger problem when a neural network has too great a number of parameters in comparison with the amount of data, data dimension and density. We can get some idea of why this effect occurs from the example of approximating function $g : R \to R$ by a polynomial on the basis of a limited number of pairs $[x, g(x)]$. If we try to interpolate the data with a polynomial of adequate degree, we can often approximate the relationship between the data points. If on the other hand we interpolate the data with a polynomial of too large a degree, then this polynomial will adapt too well to our training dataset and will completely miss the real relationship by which the data points were generated (3.5). A similar thing happens if a network has too small a number of parameters. A neural network, or a polynomial in the graph below, is not able to handle the complexity of the relationship between the input data points.

Overfitting also increases with the length of training of the network as can be seen on Figure 3.6. While training a network we can monitor the overfitting by plotting the loss both on the training set and on another set of data, which has not yet been shown to the network. This set should be different from the one on which we run the final evaluation.

3.3 Advanced concepts

Having explained how a simple neural network looks and how the training algorithm works, it is still necessary to explain several important details which justify the high-level structure of neural networks and to introduce two more advanced concepts used in our experiments.

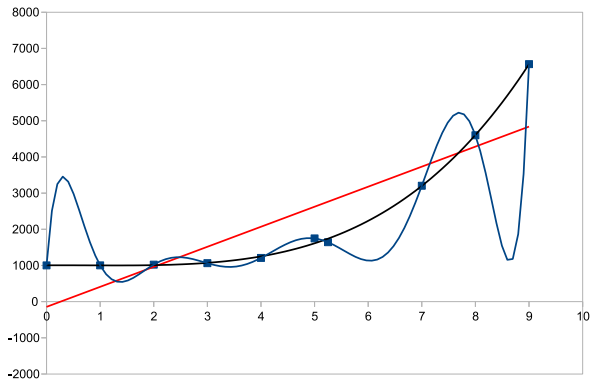

Figure 3.5: Overfitting shown on example of polynomials. If we are fitting the data with a polynomial of too small a degree(red), it is unable to describe the trend of data. If we on the other hand fit the data with a polynomial of too big a degree(blue) it may adapt too well to our training data missing the real relationship. If we use a polynomial of a proper degree (black) it approximates the real trend well.

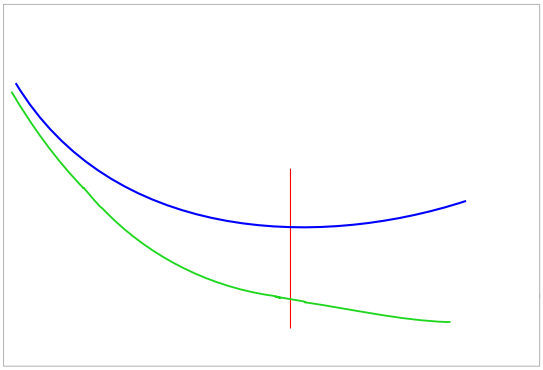

Figure 3.6: The graph shows a schematic comparison of the losses on the training(green) and testing(blue) sets. We can see that up to a certain point (marked by a red line) both losses decrease – the network is learning to solve the problem. After this point, the training set loss decreases further, while the loss on the testing set increases – the network has started to overfit, i.e. to adapt too much to the specifics of the training set.

3.3.1 Universal approximation theorem

As mentioned, a single perceptron is not strong enough to accurately approximate the vast majority of functions; we have therefore used the multi-layer perceptron (MLP) architecture. But how powerful is this architecture? The ability of MLP to approximate functions depends on its activation function. If the activation function is linear, the whole MLP $f : R^n \to R^m$ will degenerate to a group of m linear classifiers. On the other hand, for proper activation functions (for instance Sigmoid, TanH and ReLU, as shown in figures 3.1,3.2) it can be proven that a neural network with n inputs, m outputs and a single hidden layer containing a sufficient number of neurons can approximate any continuous function with arbitrary precision. This theorem is proven for Sigmoid and TanH by the universal approximation theorem Hornik [1991] and by a different theorem for ReLU Sonoda and Murata [2015]. Those theorems show that MLP is sufficiently powerful. Unfortunately, they do not help in any way with the construction of such a network. The proofs are based on direct construction of the network on the basis of knowledge of almost the whole function which we want to approximate. In practice we have the value of this function only for a very limited number of points, so we cannot follow the procedure described in the proof.

Even though a sufficiently large network with a single hidden layer can approximate arbitrary continuous function arbitrarily well, using a network with only one hidden layer may not always be ideal: there exist functions which can also be approximated by a network with one hidden layer, but if we decide to approximate them with such a network, we have to increase the size of the layer exponentially (in terms of number of layers) in comparison with a network which has multiple hidden layers and approximates the same function equally well.

3.3.2 Training of ReLU

As mentioned, if we want to train a neural network we have to first differentiate it. But looking more closely at activation function ReLU, defined as $ReLU(x) := max(0, x)$, we find that it has no derivative for x=0. What if the derivative is required at this point? A commonly used method is to replace the derivative in zero by either a derivative from the left in zero (0) or by a derivative from the right in zero (1). It is argued [Goodfellow et al., 2016, p.192] that such an approach is not problematic, for exactly the same reason a computer's rounding errors during training and inference are not problematic: in this case, we can imagine we have simply shifted the value of x from 0 to $0 + \epsilon$ or $0 - \epsilon$, $\epsilon > 0$ before computing the derivative. We can imagine that the epsilon is much smaller than the smallest positive number our floating-point data type can represent. Therefore the effect of our modification should be much smaller than the effect of normal rounding done by computer [Goodfellow et al., 2016, p. 192]. The same approach can be adopted for computation of the derivative of the hinge loss when $f(x) = 1$.

It can also be seen that for $x < 0$ derivatives of ReLU are zero, which means that during certain training steps in a certain state of the network, and with a certain input, we may be unable to modify the weights of a neuron containing the ReLU activation function. This is usually not an issue, especially since the training process is carried out on batches of data. Usually, there are enough inputs in which this neuron has non-zero output value and therefore its weights

can be modified. In addition, even if the neuron returns 0 for all inputs from the training set, there is still a chance that other neurons from the same layer will cause such a change in the rest of the network that this neuron will start to receive different input values and so start producing non-zero outputs. Unfortunately, if this does not happen our network will contain several neurons which will never be activated.

An even bigger problem appears when a larger number of neurons reach such a state: if one complete layer reaches this state, we will lose the ability to train the network. We may be risking this problem for instance when we use too high a learning rate, or when we are pushing some of the outputs of the network too far towards zero.

ReLU also has another interesting property. As we have mentioned above, if the activation function is linear, the neural network has only a very limited ability to approximate functions. Therefore it might be seen as surprising that ReLU, a function which is composed of two linear parts, is "nonlinear enough" for a universal approximation theorem to hold for it too, and to behave well in practice.

Why do we use ReLU despite knowing it suffers from serious problems? ReLU started to be used in very deep neural networks (ca. 100 layers) because other activation functions have problems with propagation of gradient in very deep networks. If we calculate the value of the gradient for a given weight of a neural network with n layers via the chain rule, we can see that its computation requires multiplication of up to n derivatives of the activation function. If the value of derivation of the activation function is smaller than one at many points, the product can become very small and the limited precision of our floating-point data type can mean large value changes due to rounding errors; the gradient can also vanish completely. Thus, we do not know how to update the weights and so will have problems training the network, or be unable to train it at all. ReLU addresses this problem because its derivative is either zero or one.

Another property of ReLU which may sometimes be useful is that it can be trained with the same speed in all the places where it has a non-zero derivative. On the other hand, it has to pay the price of having no upper bound for output values. We can compare this behaviour with TanH, which is trained faster when its value is around 0 and slower with larger or smaller values. During the training process, samples from the training batch may cause the value which goes into the TanH function to be very close to -1. If in this state we find that we need to increase the values which flow into the neuron, we have to undertake a large number of small steps, because the derivatives of Sigmoid are small. This is not an issue with ReLU and therefore its speed of convergence in some problems can be faster (even several times faster) than that of TanH Krizhevsky et al. [2012]. On the other hand, TanH has bounded output value, which also has some positives.

3.3.3 Dropout

The networks discussed up to now have had static structure. However, it has been shown Srivastava et al. [2014] that a type of random temporary structure change called dropout can lead to improved results. If we have a layer with dropout, then during each iteration of the training process we select each neuron

of that layer only with a given probability (usually 0.5 for hidden layers and 0.8 for the input layer). In cases where a neuron is selected, we work with it as with a normal neuron present in a normal layer. We compute its values and train it normally. In cases where a neuron is not selected, we behave as if it was not in the network at all. During inference we use all the neurons of the layer. We simply multiply the output of each of the neurons by the probability that the neuron will be selected during the training phase, so that the input values of the consequent layer will not become too large. Usage of dropout can lead to the creation of more robust models and decrease the risk of overfitting Srivastava et al. [2014]. We can look at a dropout network as if we were training an exponentially large number of models with shared weights and then averaging those models Goodfellow et al. [2013]. This process shows some similarity with bagging, a general machine learning technique where we train multiple models with independent parameters on different subsets of the training set and then use some combination of their outputs during inference [Goodfellow et al., 2016, p.258].

3.3.4 Maxout

Usually, the output value of a neural network is the output value of the neuron in the last layer. However, network output can be defined also in a different way. We can have multiple neurons in the last layer and select only the one with maximal activation as the output of the network. We then propagate the prediction error only through this neuron while ignoring the other neurons of the last layer. This method is called maxout and has yielded interesting results on some problems Goodfellow et al. [2013]. In order to be able to train a maxout network well, we have to combine it with dropout. Otherwise, the last layer will be dominated by just a few neurons, one of which will always have higher activation than the remaining neurons, thus effectively preventing the training of those remaining neurons.

3.4 Representation learning

Up to now we have discussed only one possible way of using neural networks, in cases where the network is presented with some input and then shown the desired result. But there exists also a completely different approach: We can train a neural network in such a way that it's output vectors will gain some special properties. While training we will not care about the exact values of the output vectors. We will only be interested in them having those properties.

For instance, we may want to create a network which transforms input data into a representation where the Euclidean distance between two data points corresponds to some form of similarity in the original data.

From this point of view, the last layer of a classifier network can be seen as a linear classifier seeking to linearly separate data, while the rest of the network tries to learn to transform data so that they become linearly separable.

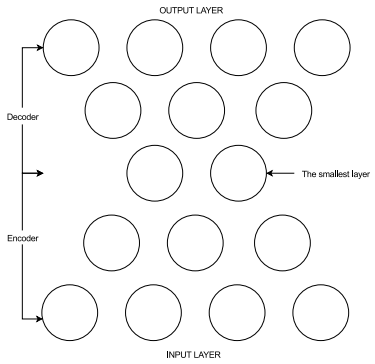

Figure 3.7: Autoencoder. During the training phase we train the autoencoder to produce the same values on the output layer as it was given on input. During the inference time we just use the part of the network labeled as encoder to extract the features.

3.4.1 Autoencoder

The autoencoder (3.7) is a common example of a network which learns representation. Its goal is to learn to compress the input data into a vector of smaller dimensions and then to reconstruct the original input from its values. The autoencoder is constructed in such a way that it is a normal $R^m \rightarrow R^m$ neural network with two special properties. The first is that the loss function penalizes only the difference between the input and the output of the network. It wants the network to learn the identity function. The second is that the network limits the amount of information which can flow through it. The layer which limits the flow the most (usually the middle one) is the layer from which we will later extract the representation. A common way to limit the flow of information through the layer is to make the number of its neurons smaller than the length of the input data. Another frequently used method is to add random noise to the values going through this layer. Since the amount of information which can flow through the middle layer is limited, the network will have to learn how to compress the information before it reaches it, as well as how to decompress it back. Since only a certain proportion of information can go through the middle layer, the autoencoder will try to select the most important information. Therefore, the part of the autoencoder between the input and the middle layer will become an extractor of the most important features.

Unfortunately, the term "most important" is not necessarily related to our needs. It relates rather to the loss function. The compressed representation will contain information whose absence would cause the greatest error. It can happen that it will not contain the information of interest, because its absence causes only a small difference between the input and output and therefore the autoencoder considers the information unimportant, instead deciding to compress something different.

To make the autoencoder extract the information we consider important, we sometimes have to modify the loss function to better reflect our needs. For this we can use so-called Siamese networks.

3.4.2 Siamese networks

Siamese networks are based on the idea that we can connect two or more different neural networks in such a way that these networks share part of their weights. The resulting network can have multiple independent inputs and outputs and its loss function is composed of multiple parts.

An example of such a network is a combination of autoencoder with classifier, when seeking to reduce the dimensions of data in such a way that the compressed representation will still contain the information which helps us to distinguish between classes.

The loss function can then be defined for example as
$loss = \alpha * (classifier(encoder(x)) - y^*)^2 + (1 - \alpha)(decoder(encoder(x)) - x)^2$,
where α is a hyperparameter which determines how much information the network will store on the basis of the variability it causes in the original data, as compared to how much it will store based on the extent to which a given piece of information helps to distinguish the class of a given sample.

Another possible use of Siamese architecture is in a network which learns to create a representation where the Euclidean distance between the outputs for two inputs corresponds to some form of similarity in the original data. One example of such a network is Google FaceNet Schroff et al. [2015], which learns to extract a vector describing a person's face from its photographic representation. The goal is to make the distance between outputs correspond to a general similarity between the people in the photos, so that we can say that if the distance between two photos is below a certain threshold it is likely that they both contain the face of a single person. This network is taught via a method called triplet loss, where in each training step the network is shown three photos A_1, A_2, B. The first two photos show the same person and the third photo someone different.
The loss function is defined as:
$loss = (f(A_1) - f(A_2))^2 - (f(A_2) - f(B))^2$.
The loss function makes the network move its outputs for photos of the same person closer to one another, and push its output for a photo of somebody else further away.

3.5 TensorFlow

For training and evaluation of our experiments we used the TensorFlow framework. This is an open-source framework developed by Google whose main purpose is to allow people to run mathematical computations efficiently with a minimum of work by allowing them to split the description of the computation from the actual execution.

With their commands users do not run the actual computations but instead only create a graph, where each node represents either a source of data or a computation; from each computational node there are labelled edges to vertices containing the input values of the computation. When a user completes the creation of a graph, she says which node she wants to evaluate and provides the data required for the computation. The TensorFlow framework finds which nodes of the graph have to be evaluated in order to compute the desired result, finds an efficient order of computation and computes the result.

This approach allows users to describe experiments in a high-level language such as Python while not losing any performance, because all the computations are done by a highly optimized code making use of parallelism and GPUs with the support of CUDA technology.

Since TensorFlow includes implementation of almost all the techniques used to train neural networks and allows implementation of the rest, it has become widely used in the field of machine learning.

4. Classification

The first problem is how to distinguish whether a given sample of communication contains activity generated by a malicious software (positive samples) or was generated purely by activity of benign software (negative samples).

This problem belongs to a group of problems called classification problems. Classification can be defined as the task of distinguishing which class a previously unseen item belongs to, when we are given a set of examples for each class.

There are three basic approaches to classification. Either we can create only two classes (benign and malign), or we can create a higher, but constant, number of classes, or we can create a multi-class classifier which classifies data into a previously unknown number of classes (e.g. a separate class for each class of malware and several classes for benign communication). One example of the last type is Google FaceNet Schroff et al. [2015], as mentioned in Chapter 3.

The last problem is generally the most interesting, but also the most difficult because it requires a large amount of data, which also have to contain more information about each sample.

Because we have no further information about the nature of each sample except whether or not it contains malicious communication, we will study only the first problem, although we try to group similarly behaving malware via unsupervised means in the chapter on similarity search.

4.1 Evaluation

Before we can start experimenting, we have to specify the method of evaluation of the results. Since the considered application of this classifier is in large companies with a large number of computers, it is crucial that the classifier has a very small false alarm rate. No business would install a system which produces a false alarm every few minutes.

Before continuing with the description of the problem it is useful to mention two terms frequently used in machine learning. The first is recall, which is the probability that a positive sample will be classified as positive. The second is precision, which is the probability that a sample classified as positive really is positive. Recall and precision go against one another. We can have a high recall and recognize almost all the positive samples, but then we will have small precision, i.e. many false alarms. Or we can have a high precision, meaning a very small number of false alarms, but then we will also have a low recall, meaning only a small proportion of malware will be detected.

In this task the customers require extremely high precision. To achieve this, we have to sacrifice the recall. From this motivation arose the measure known as FP-50, which is regularly used in the field of malware detection via machine learning in the field of network security Kohout and Pevny [2015]. The authors of this measure hold the opinion that a recall of just 50% is enough, because usually the malware will communicate with its control servers repeatedly yielding many chances to detect it. Therefore, they defined FP-50 as follows:
$FP - 50 = \frac{1}{C} * |\{c | f(c) >= median(f(m), m \in M), c \in C\}|$,
where C is a set of clean samples and M is a set of malicious samples.

If we take the median classifier output for malicious samples, then FP-50 is the percentage of background samples for which classifier output is higher than this median.

The rationale behind this is that in real-world application we will take this median and use it as a decision boundary of our classifier. If confidence that a sample is positive is at least as great as the median, we mark that sample as positive. Otherwise it is marked as negative. This approach will yield a recall of 50% and a false alarm rate equal to FP-50.

4.2 Experiments

The goal here is to design a neural network which can be trained to distinguish between samples of communication containing malware (positive samples) and samples of communication generated by benign traffic (negative samples).

A general approach to create a neural network for the purposes of classification is to take a classified sample x as the network's input, then have several hidden layers and on top of this architecture place one output neuron of output value y, corresponding to a degree of confidence that a given sample is positive. The network will be trained by minimization of difference between the output y and the desired output y^*. We experimented with loss functions MSE and hinge loss, both described in Chapter 3.

The use of hinge loss is important for training classifiers with ReLU as the activation function of their last neurons (networks with maxout). Let us label each negative sample with 0 and each positive sample with 1. Then, the MSE will penalize the network for returning a value larger than 1 even if the sample is positive. The training process will then try to reduce the network's output for the malicious sample, which is in direct opposition to our needs.

The layers of the networks are fully connected – each neuron of the next layer is connected to each neuron of the previous layer. In the case of GMM data, this design can be justified by the fact that the spatial structure of GMM data does not contain any patterns we can exploit. If the histogram data had been structured as "hard" histograms we might have considered a four-dimensional convolutional layer with one element of stride 1 and shape 3X3X3X3, or similar, to help the neural network train an architecture able to handle the noise created by rounding. But since our histograms are created as "soft" this was not necessary, and there was no other spatial structure of the input we could exploit.

We tested two basic designs of the output layer – either the last layer contained only one neuron, or it contained multiple neurons and worked as a maxout layer.

The resulting networks were trained on 3 mil. background training samples and 2,800 infected training samples in the case of GMM data, and on 10 mil. background and 2,800 infected training samples in the case of soft histogram data. An increase in the number of background samples was tested on a small set of representatively chosen architectures and did not lead to an observable improvement of results. This result is not very surprising, because the huge disproportion between the number of clean and infected samples tells us that our performance is most likely limited by the small number of infected samples.

Because of the limitedness of our set of positive samples (2,800 in the first week of communication and 1,578 in the second week), no validation set is used.

Architecture	average FP-50
30 ReLU X 1 Sigmoid	0.00145
100 ReLU X 1 Sigmoid	9.68e-005
200 ReLU X 1 Sigmoid	6.07e-005
300 ReLU X 1 Sigmoid	5.70e-005
500 ReLU X 1 Sigmoid	5.38e-005
800 ReLU X 1 Sigmoid	5.52e-005

Figure 4.1: FP-50 of networks trained on GMM 600 via Mean Square Error.

We only use the first week as the training set and the second week as the testing set.

This has important consequences. If we based our choices of the stopping time, learning rate decay and architecture too much on the performance of our other networks on the testing set, we would be at risk of accidentally tuning our models to work well with the testing set. Which would cause us to overestimate the performance of our models in evaluation.

To minimize this risk, fixed learning rate decay is used with a fixed number of training steps, instead of guiding the training by comparing the loss on the training dataset with the loss on a different dataset.

The results were validated by training each model twice with the same set of malware samples and a different set of background samples. The results were then averaged.

4.2.1 GMM600

In this section we describe our experiments on a GMM600 dataset. First of all we tried basic networks with one hidden layer (4.1). We see that architectures with 200 neurons and more provide interesting results.

During the experimental phase we also wanted to try networks with maxout activation. At first we tried to train such networks with MSE, penalizing the network via loss function
$loss = (f(x) - y^*)^2$
where y^* is the desired label (0 for a negative sample and 1 for a positive sample), x is the input vector and f is the function computed by a neural network. As can be seen in the first two columns of 4.2 the networks did not achieve good results. To find the cause, we tried taking the average of the neuron activations in the last layer of each network as the output of the network. Even though the results improved (see the third column of the same table), they remained unsatisfactory, especially because of the overall instability of the training.

It transpired that the loss function used for the training was incorrect. The output of the maxout layer with ReLU neurons is potentially unlimited. This means that the output of this layer for a positive sample can exceed 1. When this happens, the MSE loss function starts to behave badly, considering the output value of the network to be too large. It therefore starts to push it down, towards one. Unfortunately, this is the opposite of what we want our loss function to do – namely to always force the network to increase its output for positive samples and decrease it for the negative ones. Thus, the loss function was effectively telling

Architecture	average FP-50	average FP-50 (average of outputs)
20 X 20 Maxout	0.00085	0.000190
50 X 50 Maxout	0.00062	7.38e-05
100 X 100 Maxout	0.00197	0.000170
200 X 200 Maxout	0.00316	7.23e-05
300 X 300 Maxout	0.00164	3.94e-05
400 X 400 Maxout	0.00172	8.83e-05
500 X 500 Maxout	0.00304	0.00141
700 X 700 Maxout	0.00163	0.00145
800 X 800 Maxout	0.00165	0.000107

Figure 4.2: FP-50 of Maxout architectures trained on GMM 600 via Mean Square Error. Average of outputs means that instead of taking the highest activity of a neuron in the last layer as the activation of the network we used the average of activations of all the neurons in the last layer.

Architecture	average FP-50
30 ReLU X 30 ReLU Maxout	4.26e-05
100 ReLU X 100 ReLU Maxout	3.59e-05
200 ReLU X 200 ReLU Maxout	3.49e-05
300 ReLU X 300 ReLU Maxout	3.57e-05
500 ReLU X 500 ReLU Maxout	3.28e-05
800 ReLU X 800 ReLU Maxout	3.18e-05

Figure 4.3: Networks with one hidden layer trained on GMM600 with hinge loss.

the network not to detect the positive samples.

To test whether this was the cause of our problems, we trained the same networks with a loss function hinge loss defined as:
$HingeLoss = max(0, 1 - f(x) * y^*)$,
where x is the input vector, f the function computed by the neural network and $y^* \in -1, 1$ the desired class (-1 for negative, and 1 for positive samples). As we can see, this loss does not suffer from the described problem. The results achieved by the training with this loss function are shown in table 4.3. As can be seen, the networks achieved better results than the same architectures trained with MSE loss function, indeed even better than the networks with one hidden layer.

We then tried models with more hidden layers (two and three), but even with only two hidden layers the networks did not train well and did not provide good results. This was likely caused by the increase in the descriptive power of the model, which allowed the models to overfit or to degenerate into a state where f(x)=0.

To overcome this problem we used L2 weight normalization. This type of normalization penalizes the model for having large values of weights by adding the term $\alpha * \sum_{i=0}^{n} w_i^2$ to the loss function. Unfortunately, there is always a drawback connected with this form of normalization. The bigger the value of α, the more a neural network can be optimized to have small weights instead of small prediction error. But the smaller the value of α, the more prone to overfitting the model. This is an example of the variance vs. bias trade-off, which is a problem commonly

Model	$\alpha = 0.1$	$\alpha = 0.01$	$\alpha = 0.001$	$\alpha = 0$
30X30X30 Maxout	0.000240	0.000110	0.000070	0.000075
100X100X100 Maxout	0.000075	0.000365	0.000055	0.50
200X200X200 Maxout	0.000640	0.000590	0.000220	1
300X300X300 Maxout	0.000550	0.000450	0.000160	1
500X500X500 Maxout	0.000460	0.000285	0.000065	1
30X30X30X1 Sigmoid	1	0.500	0.500	1
100X100X100X1 Sigmoid	0.001890	0.006750	0.007720	1
200X200X200X1 Sigmoid	0.005120	0.007910	0.006800	1
300X300X300X1 Sigmoid	0.005750	0.008130	0.009020	1
500X500X500X1 Sigmoid	0.005750	0.007590	0.007410	1

Figure 4.4: Networks with two hidden layers trained on GMM600 with MSE loss with weight decay α.

Architecture	average FP-50
30 ReLU X 1 Sigmoid	0.00393
100 ReLU X 1 Sigmoid	0.00394
200 ReLU X 1 Sigmoid	0.00390
300 ReLU X 1 Sigmoid	0.00365
500 ReLU X 1 Sigmoid	0.00355
800 ReLU X 1 Sigmoid	0.00348

Figure 4.5: Networks trained on GMM600_480 with MSE loss.

faced in machine learning.

The results of our measurements are shown in table 4.4.

From the results we can notice several things.

Firstly, the L2 weight normalization helped us to train deeper models. Unfortunately, even with this normalization, use of deeper models did not lead to any improvement in results.

Secondly, once again, it can be seen that the presence of the maxout output layer leads to better results.

4.2.2 GMM600_480

This section describes our experiments with the GMM600_480 dataset. Since the structure of the GMM600_480 data resembles that of the GMM600, similar architectures were investigated.

First we tried simple networks with one hidden layer (4.5).

We observed that their training was stable but for unknown reasons their performance was much worse than the performance of the same architectures on GMM600.

Then we trained models with maxout 4.6. After the experience with GMM600, hinge loss was used.

The maxout models performed similarly to the same architectures on GMM600 (perhaps even slightly better, but more measurements would be required to be sure). We also tried models with two hidden layers but they did not improve the results.

Architecture	average FP-50
30 ReLU X 30 ReLU Maxout	0.00306
100 ReLU X 100 ReLU Maxout	0.000209
200 ReLU X 200 ReLU Maxout	2.55e-05
300 ReLU X 300 ReLU Maxout	2.34e-05
500 ReLU X 500 ReLU Maxout	2.96e-05
800 ReLU X 800 ReLU Maxout	0.0591*

Figure 4.6: Networks trained on GMM600_480 with hinge loss. * The second model diverged and was replaced by a third model.

4.2.3 Soft histograms

The third dataset contained data aggregated in the soft histogram format. This format has a very high dimensionality of $11^4 = 14641$ bins, while being extremely sparse – the average number of non-empty bins is approximately 63 for malware samples and approximately 47 for clean samples. This combination of sparsity and high dimensionality is the root of all problems faced during the training.

It causes problems because sparse high-dimensional data force us to accept input vectors of a very high dimension but do not provide us sufficient information about the "meaning" of different components of the input vectors. In our histogram bins which have a non-zero value are found only in one, benign, data sample. How sure can we be that the value of this bin helps us distinguish between malicious and benign samples?

While inspecting our input data we found that a large number of bins of histograms are empty in the whole testing set. If they can be efficiently removed, the size of the network can be decreased, thus reducing its space requirements and required evaluation time.

In the training dataset we know that there exist non-empty bins only at coordinates (w,x,y,z), such that w $\in$ {1,...,11}, x $\in$ {1,...,11}, y $\in$ {0,1,...,5} and w $\in$ {0,1,...,6}. By completely removing the bin coordinates which are outside of those boundaries, we can effectively reduce the dimensions of our data to 10*10*5*7 = 3,500 dimensions. We thus reduce the number of weights in the first layer of our neural network by 76%. This is positive, because the first layer is likely to use the majority of the memory and computational power required by the whole network. If during the testing phase a non-zero value falls into one of the removed bins, we behave as if the value was not present at all. This corresponds to the behaviour we expect from neural networks in such a case.

Of the remaining 3,500 bins, 1,317 were empty in all the samples from the training set. We decided not to remove them, because they are spread all over the histogram, which complicates their removal.

We began by training models with one hidden layer.

As we can see, both types of models show unsatisfactory behaviour. To check whether this was caused by overfitting of the first layer to the training data, we tried to add additional dropout regularization, but unfortunately Fig. 4.9 shows that this did not lead to any improvement.

The architectures 4.9,4.10 were trained with both MSE and hinge loss functions. While the FP-50 scores are almost the same, the MSE models surprisingly have better precision-recall curves (4.11).

Architecture	average FP-50
30 ReLU X 1 Sigmoid	0.00677
100 ReLU X 1 Sigmoid	0.00523
200 ReLU X 1 Sigmoid	0.00468
400 ReLU X 1 Sigmoid	0.00327
800 ReLU X 1 Sigmoid	0.00178
1000 ReLU X 1 Sigmoid	0.00185

Figure 4.7: Networks trained on Soft histograms with MSE.

Architecture	average FP-50
30 ReLU X 30 ReLU Maxout	0.00776
100 ReLU X 100 ReLU Maxout	0.00481
200 ReLU X 200 ReLU Maxout	0.00352
400 ReLU X 400 ReLU Maxout	0.00312
800 ReLU X 800 ReLU Maxout	0.00264
1000 ReLU X 1000 ReLU Maxout	0.00263

Figure 4.8: Networks trained on Soft histograms with MSE. Hinge loss was tested and provided similar results.

Architecture	average FP-50
100 ReLU Dropout X 100 ReLU Maxout	0.00622
200 ReLU Dropout X 200 ReLU Maxout	0.00483
400 ReLU Dropout X 400 ReLU Maxout	0.00336
800 ReLU Dropout X 800 ReLU Maxout	0.00294
1000 ReLU Dropout X 1000 ReLU Maxout	0.00266

Figure 4.9: Networks trained on Soft histograms with MSE loss.

Architecture	average FP-50
100 ReLU Dropout X 100 ReLU Maxout	0.00613
200 ReLU Dropout X 200 ReLU Maxout	0.00460
400 ReLU Dropout X 400 ReLU Maxout	0.00357
800 ReLU Dropout X 800 ReLU Maxout	0.00282
1000 ReLU Dropout X 1000 ReLU Maxout	0.00277

Figure 4.10: Networks trained on Soft histograms with hinge loss.

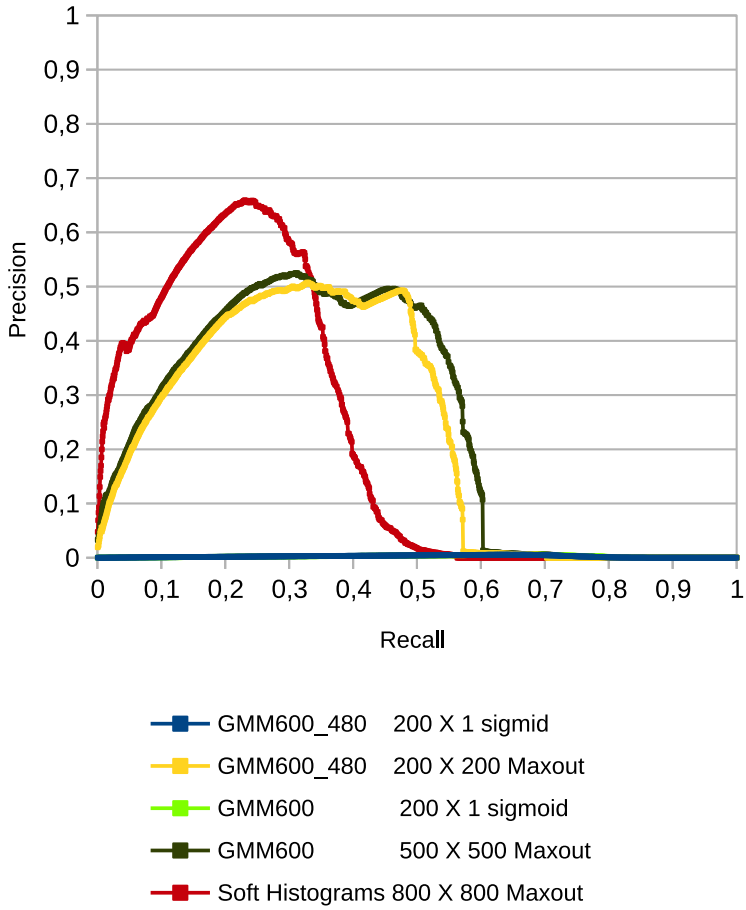

Figure 4.11: A graph of precision-recall curves of selected architectures. We can notice that classifiers with Sigmoid activation function show poor performance. Networks trained on Soft Histograms show better PR curves when trained with MSE loss.

Training models using the soft histogram data were generally more complicated than those using GMMs, and had worse results.

4.3 Evaluation

Generally speaking, we can conclude that neural networks with one hidden layer and maxout activation did not produce worse FP-50 than the other neural networks, and that neural networks performed better with GMM data than soft histograms. When we analyze the precision-recall graph 4.11 we can notice that networks with output layer with sigmoid show worse performance than we would expect from it's FP-50. We can also see that classifier trained on Soft histogram data is in fact relatively good. We would just have to sacrifice more recall to reduce it's false alarm rate.

We see that the best performance achieved on GMM data was around FP-50 3e-05. But what does this value mean in practice? If a model has an FP-50 equal to 3.20e-05, while catching half of the malicious communication, it causes one false alarm per 33,333 inspected samples. Since each sample contains five minutes of

communication of one host, this means that one false alarm is caused per 347 man-days of network activity (considering eight-hour working days). This result is promising. It shows that neural networks can learn to distinguish between malicious and benign communication with a high level of accuracy even when analysing encrypted traffic and relying purely on metadata. However, future improvements will be necessary to make this method practical, because one false alarm every day is unacceptable in a company with just 300 employees. Because the number of positive samples is extremely low, we expect that the necessary improvement can be reached purely by increasing it.

If we compare our results with the yet unpublished results of our colleagues Přemysl Čech and Tomáš Komárek, Kohout et al. [2017] using different machine learning methods on the same task, we can conclude that the results for neural networks are slightly worse, but still comparable with the results of random forests and better than the results of k nearest neighbours. It can also be seen that GMM data aggregation leads to an improvement in the performance of all machine learning algorithms.

5. Similarity Search

The second problem studied is similarity search. If an infected computer is found in a network, it is important to check whether there are any other computers infected with the same or a similar virus. Since a computer infection may also have been detected by a method other than our classification algorithm, this algorithm should work also for samples which are below the decision boundary of our classifiers.

To achieve this we would like to create a distance function $d(x, y)$ from pairs of communication descriptors into R such that the following two inequalities hold with a high probability:
(1)$d(m_1, m_2) < d(m_1, n)$
(2)$d(m_1, m_2) < d(m_1, c_1)$
where m_1 and m_2 are two distinct samples of communication of one piece of malware from distinct computers, n is a sample of communication of a different piece of malware and c_1 are samples of clean communication.

These inequalities imply that if we take a sample m of malware communication and order all the remaining samples of communication from our database y by the value of $d(m, y)$ in ascending order, then if there exists a different sample of communication of the same malware, it will be in one of the first positions of this ordering. It would also be interesting if inequality
(3)$d(m_1, n) < d(m_1, c_1)$
held with a non-negligible probability, since if we inspect the similarity search results, it is better if we find a computer infected with a different virus than the one we were looking for than if we find a computer which is not infected at all.

Below we will discuss the construction of d() via the means of neural networks, but first we have to show that having such a function really solves our problem.

With a function $d()$ for which the first two inequalities hold, for use in a real-world scenario we will first have to find M_{inf} – a sample of communication of the infected computer that contains some communication of the virus. We must then compute $d(M_{inf}, x)$ for all the communication samples x of different computers. Then we can select which samples are closest to M_{inf} but come from a different computer.

After the search, we can consider filtering the results by selecting only neighbours with a distance from M_{inf} smaller than a certain threshold, and not showing the samples which are too far from M_{inf} to be infected by the same virus.

Theoretically, this will provide a list of the computers most probably infected by the same virus.

How does a user find a M_{inf} sample? The similarity search should be used after an administrator recognizes that a given computer is infected. She can recognize this in either of the following ways. Either she is warned by our classification system, whereupon she will be given a sample of infected communication by that system, or she recognizes the infection by different means (e.g. after being warned by antivirus software on the computer, or noticing visible behaviour of the malware). Even in the second case, there is still a chance that the administrator will be able to obtain an infected sample of communication. If she is not able to do so, she must perform the search on all the samples of communication from

that computer which were created within a certain period of time, as if each of them was infected. Unfortunately, in this case the results of nearest neighbour search may also include clean computers which have produced a certain clean activity C, which is close to the clean activity C' of the infected computer at a time when the virus was not communicating.

5.1 Implementation

This section will discuss how to create the mapping $d()$.

The first way to implement $d()$ is to create a neural network which takes two input vectors, descriptors of communication, and produces one number, the similarity of the communications.

This is disadvantageous because such a function would scale badly. For example, a search for communications similar to one of k samples of communication of infected computers within a database of n communication samples would require $n*k$ evaluations of function d, and we would have to work with each of n samples in the database. With a huge database this may easily be too much. It may also prove difficult to train such a network with our limited number of positive samples, because this architecture is likely to require a much greater number of weights to work.

Instead, we can take inspiration from the structure of Google FaceNET Schroff et al. [2015], which is trained to compare photos of faces and say whether they contain the same person. This network does not implement the function d() directly, but instead implements function f(), which takes one photo and computes a vector of features such that the Euclidean distance between f()'s outputs for two faces corresponds to their similarity. It basically implements the function $d(x, y)$ as $d(x, y) = ||f(x) - f(y)||_2$, and we will do the same.

Google FaceNET was trained by enforcing
$||f(x_1) - f(x_2)|| + \alpha < ||f(x_1) - f(y)||$, where x_1,x_2 are images from the same class (belonging to the same person) and y is from a different class, through minimization of
$\sum_{i=1}^{N}[||f(a_i) - f(b_i)||_2^2 - ||f(a_i) - f(c_i)||_2^2 + \alpha]_+$
where a_i and b_i are photos of faces of a single person, c_i is a face of a different person and $[x]_+$ means that we consider x only when $x > 0$.

This approach is clearly much more efficient. If we want to run similarity search requests on a database we may consider storing the data in the form of transformed descriptors $f(x)$. Then we need to compute the function $f()$ only once for each sample. While searching the database we may also take advantage of the distance satisfying the metric axioms, which may help us to further speed up our search Yianilos [1993].

To train the network, ideal would be the use of some modification which forces the mapping f() both to project similar types of malware close to one another and to project all clean samples far from all malware samples.

The present case is more difficult because of the small number of malware samples, and because we cannot apply the Google FaceNET method directly, since our data do not contain any information about which infected communication sample contains what kind of malware. Therefore, we have to change our goal to training a network which enforces only the third equation (a malware sample is

closer to a malware sample than to a clean sample) and tries to enforce the first one (similar pieces of malware are closer to one another than to other malware) purely via unsupervised learning methods.

We will also try to exploit the possibility that the second-but-last layer of classifiers trained in a supervised manner (with labels malware and clean) may produce a useful representation of the input data, as e.g. in Razavian et al. [2014].

Unfortunately, there is another problem in the evaluation of the results. Ideally, the similarity search would be run to check the extent to which nearest neighbours are generated by the same type of malware. Unfortunately, there is no information on which sample contains activity of which malware. Therefore, the results must be evaluated indirectly.

Because of the nature of our data we can only evaluate whether the nearest neighbours of samples of malware communication are samples of malware communication. This is very unfortunate, because with this method we are unable to distinguish between a network which creates a more advanced mapping and a network which is in fact just a masked classifier and maps all the benign samples close to one point of m-dimensional space and all the malign samples close to another point. We try to detect this behaviour from plots created by a non-deterministic dimensionality reduction technique called TSNE van der Maaten and Hinton [2008]. If we cluster the outputs of our network and see only two large clusters, one for malicious and one for benign data, we know that the described behaviour is likely to be happening. The result which does not show that the network suffers with this issue is a larger number of clusters containing each class.

When using similarity search, usage of some initial filtering of input data can be considered: the fewer input data are used, the higher the precision of the similarity search. For this, a classifier with a very high recall (ideally 100%) and as high an accuracy as possible would be required. Unfortunately, from the precision-recall curves computed for classification it seems that it would be extremely difficult to create such a classifier; therefore, we cannot use this method.

5.2 Experiments

At first we considered pure autoencoder networks and networks which are trained to just minimize the distance between positive samples and maximize distance between positive and negative samples, but both of them have theoretical disadvantages.

The representation created by the first architecture is supposed to contain compressed information about the nature of the input sample, but there is no guarantee that the Euclid distance between the created representation of two samples will carry the required meaning. There does not necessarily be anything forcing network to put malicious samples close to other malicious samples.

The second architecture can on the other hand guarantee, that positive samples will be closer to other positive samples than to negative samples but does not guarantee that it will store any information about the input but it's class. It is possible that it will just learn to classify the original samples and to project all the malicious samples close to one point of m-dimensional space and all the benign samples close to a different point, which renders similarity search impossible.

We can notice that the previously mentioned architectures are in some way complementary. This motivated us to create a siamese architecture combining both of them into a single network.

Let *encode*() denote the function computed by the part of an autoencoder preceding the layer from which we will extract the representation and *decode*() the part following this layer, m_1 and m_2 malicious samples and c clean sample. Then the loss function of our siamese architecture is defined as:
$malware\_malware\_distance = (encode(m_1) - encode(m_2))^2$
$malware\_clean\_distance = (encode(m_1) - encode(c))^2$
$autoencoder\_clean = (decode(encode(m_1)) - m_1)^2$
$autoencoder\_malware = (decode(encode(c)) - c)^2$
$loss = \alpha * (malware\_malware\_distance - malware\_clean\_distance) +$
$+ (1 - \alpha) * (autoencoder\_malware + autoencoder\_clean)$

We can see that the loss function seeks to minimize the Euclid distance between samples of malware and maximize the distance between malicious and benign samples while in the same time trying to force the representation created by the function *encode*() to retain the information about the nature of samples by minimizing the loss of autoencoders. Parameter α then balances between these two behaviours. The smaller the α is, the more the network behaves as an autoencoder and the less as the other prototypical network, and vice versa.

We have also tried to improve the resulting architecture by modification of process of selection of m_2. Normally, all the samples were selected randomly, but we realized that it may not be optimal, since it forces all the malicious samples to be close to one another. We considered it better to select m_2 in such a way that it is already close to m_1 therefore we train the network to create more clusters than just one. Unfortunately, a straightforward approach to find the closest samples of malicious communication and use one of them as m_2 is computationally expensive, because it requires us to evaluate the network on all the training samples of malware in each training round. Therefore we used only randomly selected 100 samples to find the nearest neighbours in each round. We also did not select the closest neighbour as m_2 but instead selected randomly one of the 10 nearest neighbours, because the closest one may be close enough to m_1 already.

We tested the architectures with several values of α and several sizes of layer producing the representation. We also experimented with using the ReLU activation function, which tends to produce sparser representation of the input data, as the activation function of the narrowest layer, but we had problems with output values of neurons of this layer increasing above all limits.

The results were evaluated in the way described above and compared with results of the same evaluation of the original data and the representation extracted from distinct classification networks. In this comparison the precision was computed as the average percentage of malicious samples among 10 nearest neighbors of each infected sample. For this evaluation we used all the 1578 malicious samples and 100,000 clean samples. The number of clean samples used for evaluation was selected so that we can measure the difference between different architectures.

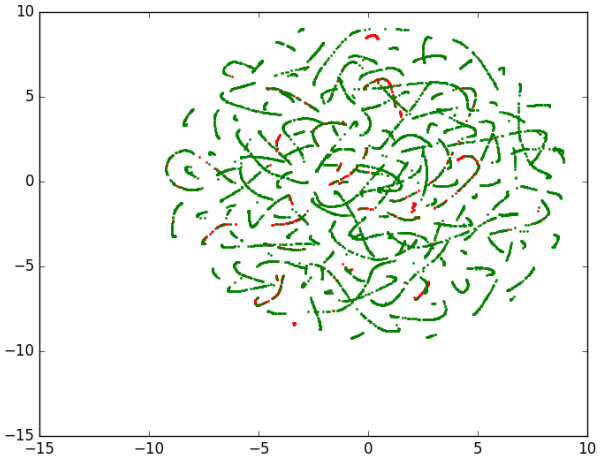

Figure 5.1: TSNE plot of a representation produced by our siamese architecture in setup with middle layer of size 400, $\alpha = 0.1$ and m_2 selected among nearest neighbors of m_1. We can see that our representation forms several small clusters. 1578 malicious samples (red) and 5000 benign samples(green).

First we show the results obtained by siamese networks:

We can see that the network with $\alpha = 0$ (5.2) (pure autoencoder) produces a completely spread set of points, while the network with $\alpha = 1$ (5.3) concentrates most of the malicious samples into two clusters. On the other hand the network with $\alpha = 0.5$ (5.3) creates several clusters of approximately equal size and therefore shows the behaviour similar to the behaviour expected from the networks we would like to train. We can also see that networks with special process of selection of m_2 show slightly better precision. The performance of such networks in clustering is comparable to the performance of standard networks.

We also discuss the results obtained by the classifier networks:

We see that the representations created by the last layer of classifiers do not form clusters of malicious data, which can be considered a hint that the more specific information about the nature of malware have been lost and therefore the output of such a network cannot be used for similarity search.

More interesting results have been shown by the representation produced by the first layer of network 300 X 1 Sigmoid 5.12, which looks 5.11 as if it was naturally forming clusters of malicious data, and also has a very high precision. The network shows results far superior to those of siamese networks, but before being sure about it's performance we have to take into account one possibility. What if the similarity search is able to detect only the "easy" samples, which would be classified as malware anyway and so the network does not add anything new to our problem, because we can just use it as a classifier instead of relying on it's performance in the similarity search?

Therefore we tested this architecture once again on similarity search. This time we used only 789 "hard" malicious samples which would normally be undetected by this network if it operated in the classifier mode. Because we decreased the number of malicious samples by one half, we decided to also decrease the number of clean samples by one half to make the problem more balanced. Under

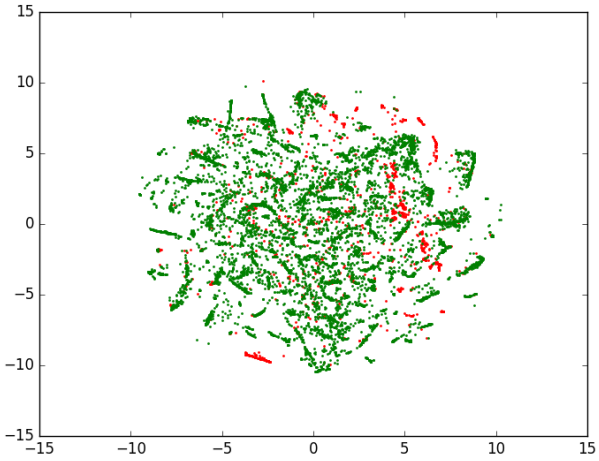

Figure 5.2: TSNE plot of a representation produced by our siamese architecture in setup with middle layer of size 100, $\alpha = 0.0$.

such conditions the precision achieved by the representation was only 0.36 which shows us that the representation of classifiers is in fact worse than the results achieved by the siamese neural networks.

For comparison we also show the performance of the original data. The precision of the original data (5.15) is comparable with or better than the results achieved by siamese networks, but the samples are mostly spread in the multidimensional space and form only a few huge clusters, which does not correspond with the properties of the representation we would like to have (5.13 and 5.14).

5.3 Evaluation

When we observe the TSNE graphs and precisions of siamese networks with α being strictly between zero and one, we can conclude that it is possible that the clusters in output representation of our siamese networks might be corresponding to groups of viruses showing similar activity, because none of the results we were able to measure is contradictory to this belief. But we still have to be careful, because the data we were given did not allow us to verify this hypothesis with 100% accuracy. Therefore further research is necessary.

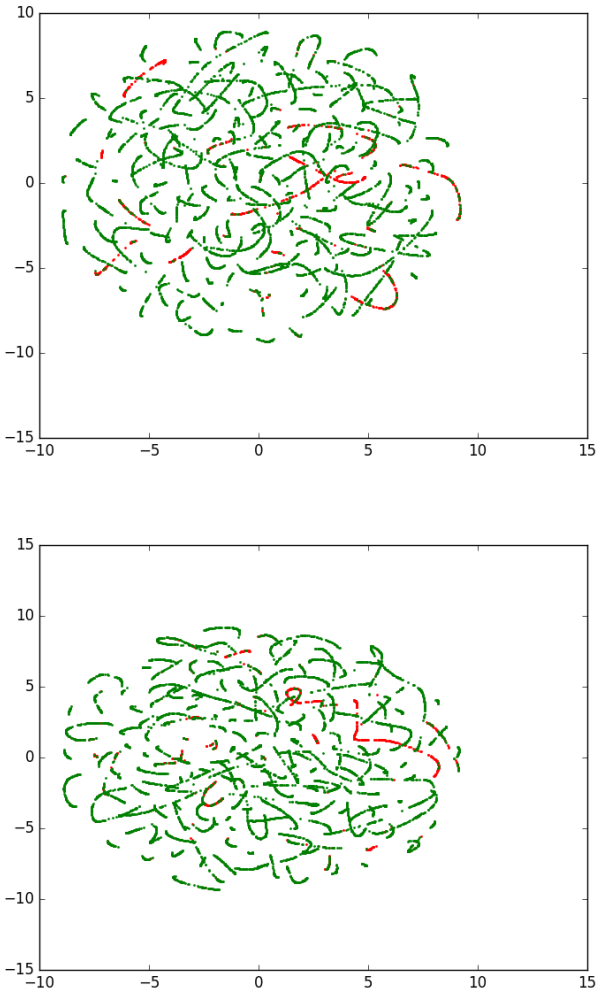

Figure 5.3: TSNE plot of a representation produced by our siamese architecture in setup with middle layer of size 400, $\alpha = 0.5$ (up) and $\alpha = 1.0$ (down). 1578 malicious samples (red) and 5000 benign samples(green).

Architecture	α	Average Precision
600X400[OUT]X600X600	0.0	0.47
600X400[OUT]X600X600	0.1	0.35
600X400[OUT]X600X600	0.35	0.39
600X400[OUT]X600X600	0.5	0.39
600X400[OUT]X600X600	0.65	0.40
600X400[OUT]X600X600	0.9	0.39
600X400[OUT]X600X600	1.0	0.38

Figure 5.4: The precision of the standard siamese architectures with sigmoidal activation function.

Architecture	α	Average Precision
600X400[OUT]X600X600	0.0	0.49
600X400[OUT]X600X600	0.1	0.36
600X400[OUT]X600X600	0.35	0.42
600X400[OUT]X600X600	0.5	0.43
600X400[OUT]X600X600	0.65	0.41
600X400[OUT]X600X600	0.9	0.41
600X400[OUT]X600X600	1.0	0.44

Figure 5.5: The precision of siamese architectures with sigmoidal activation function, employing the special process of selection of m_2 as described above.

Architecture	α	Average Precision
600X200[OUT]X600X600	0.0	0.50
600X200[OUT]X600X600	0.1	0.38
600X200[OUT]X600X600	0.35	0.40
600X200[OUT]X600X600	0.5	0.40
600X200[OUT]X600X600	0.65	0.40
600X200[OUT]X600X600	0.9	0.39
600X200[OUT]X600X600	1.0	0.42

Figure 5.6: The precision of the standard siamese architectures with sigmoidal activation function.

Architecture	α	Average Precision
600X200[OUT]X600X600	0.0	0.47
600X200[OUT]X600X600	0.1	0.42
600X200[OUT]X600X600	0.35	0.41
600X200[OUT]X600X600	0.5	0.41
600X200[OUT]X600X600	0.65	0.44
600X200[OUT]X600X600	0.9	0.39
600X200[OUT]X600X600	1.0	0.43

Figure 5.7: The precision of siamese architectures with sigmoidal activation function, employing the special process of selection of m_2 as described above.

Architecture	α	Average Precision
600X100[OUT]X600X600	0.0	0.46
600X100[OUT]X600X600	0.1	0.41
600X100[OUT]X600X600	0.35	0.39
600X100[OUT]X600X600	0.5	0.42
600X100[OUT]X600X600	0.65	0.40
600X100[OUT]X600X600	0.9	0.38
600X100[OUT]X600X600	1.0	0.42

Figure 5.8: The precision of the standard siamese architectures with sigmoidal activation function.

Architecture	α	Average Precision
600X100[OUT]X600X600	0.0	0.47
600X100[OUT]X600X600	0.1	0.43
600X100[OUT]X600X600	0.35	0.39
600X100[OUT]X600X600	0.5	0.42
600X100[OUT]X600X600	0.65	0.43
600X100[OUT]X600X600	0.9	0.40
600X100[OUT]X600X600	1.0	0.43

Figure 5.9: The precision of siamese architectures with sigmoidal activation function, employing the special process of selection of m_2 as described above.

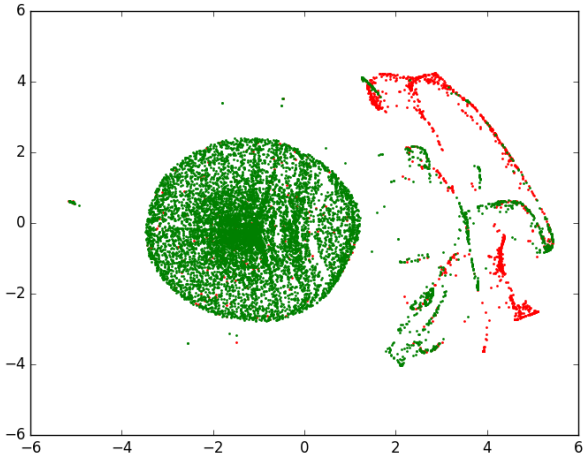

Figure 5.10: TSNE plot of data produced by the last layer of 50 X 50 Maxout classifier. We can see that this layer does not produce representation in which malicious data form clusters.

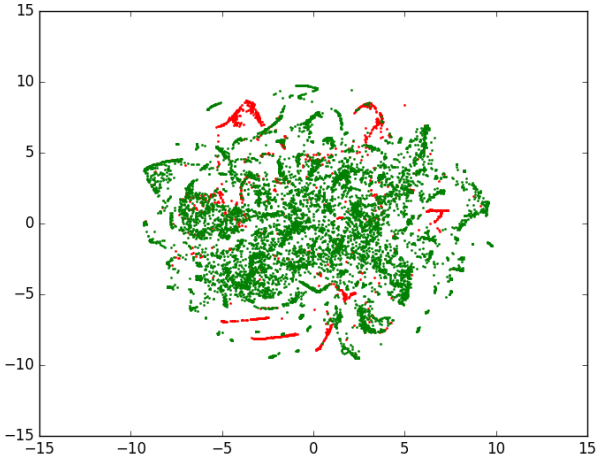

Figure 5.11: TSNE plot of data produced by the first layer of 300 X 1 sigmoid classifier. We can see that the output of this layer contains one big and a few smaller clusters of malicious data. 1578 malicious samples (red) and 5000 benign samples(green).

Architecture	Average Precision
50ReLU X 50ReLU Maxout [OUT]	0.62
800ReLU X 800 Dropout [OUT]	0.63
200ReLU X 200ReLU X 200ReLU Maxout [OUT]	0.64
200ReLU X 200ReLU[OUT] X 200ReLU Maxout	0.64
300ReLU X 1 Sigmoid	0.66

Figure 5.12: The precision of the classifier networks.

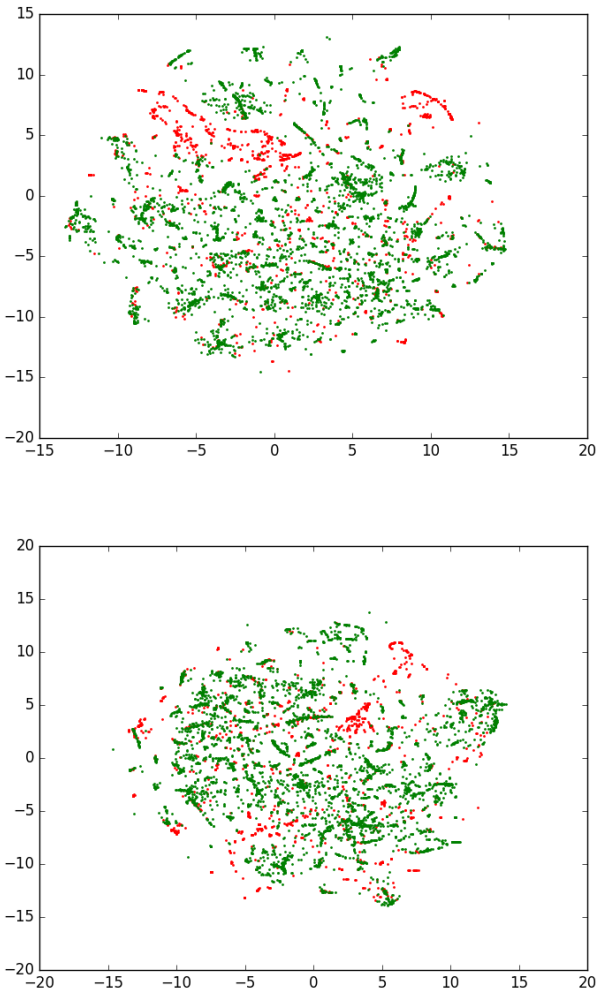

Figure 5.13: TSNE plot of GMM600 (up) and GMM600_480 (down) data. 1578 malicious samples (red) and 5000 benign samples(green).

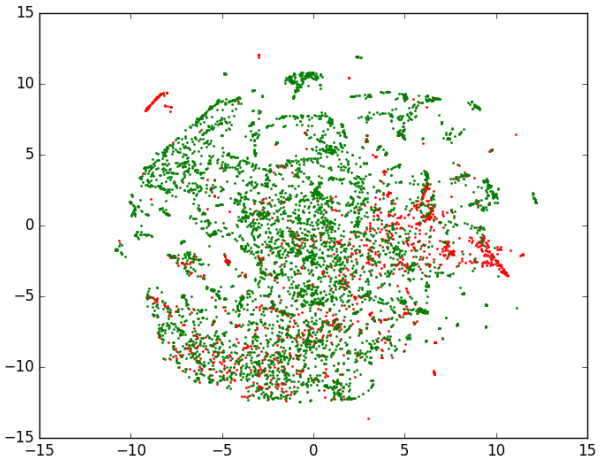

Figure 5.14: TSNE plot of the Soft histogram data. 1578 malicious samples (red) and 5000 benign samples(green).

Data	Average Precision
GMM600	0.38
GMM600_480	0.41
Soft Histograms	0.49

Figure 5.15: The top 20 precision of the original data.

6. Conclusion

This thesis has explored the possibility of detection of malware within encrypted HTTPS traffic on the basis of metadata measured and aggregated by Cisco. It has tested distinct architectures with distinct types of aggregated data. Promising results have been achieved, which once again shows that metadata contain enough information to detect malware, and that neural networks are able to make use of such information. Because networks were trained using a very small set of infected samples, we expect that the achieved results can be further improved by simply using a training dataset with a larger number of infected samples.

The achieved results were comparable with, although slightly worse than the performance achieved by random forests, and better than the results achieved by k nearest neighbours, which shows that neural networks can be efficiently used on this type of problem.

In the second part of the thesis we experimented with training of neural networks for similarity search among samples of communication. The goal was to create an architecture which groups together samples of communication containing activity of similar malware. We designed a class of architectures to help solve the problem, and tried to train and evaluate these architectures on our dataset. We achieved a partial success by creating an architecture which creates clusters of infected communication and projects the infected samples close to one another, but unfortunately our experiments were limited by the absence of more detailed information about the nature of the traffic within the used dataset. Therefore, we cannot determine whether each cluster contains only communication created by a single malware family or whether all the communication of a single malware family is within a single cluster. Similarity search requires further research on a different dataset containing information about the nature of data samples, which would allow both a more precise evaluation of our partially unsupervised architecture and testing of a multiclass classifier resembling the Google FaceNET architecture (Schroff et al. [2015], described in Chapter 3).

Bibliography

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

Ian J. Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron C. Courville, and Yoshua Bengio. Maxout networks. In *ICML (3)*, volume 28 of *JMLR Workshop and Conference Proceedings*, pages 1319–1327. JMLR.org, 2013.

Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257, 1991. doi: 10.1016/0893-6080(91)90009-t.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

Jan Kohout and Tomas Pevny. Automatic discovery of web servers hosting similar applications. *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2015. doi: 10.1109/inm.2015.7140487.

Jan Kohout, Tomas Komarek, Premysl Cech, Jan Bodnar, and Jakub Lokoc. Learning communication patterns for large discovery of https data. 2017.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems*, NIPS'12, pages 1097–1105, USA, 2012. Curran Associates Inc. URL `http://dl.acm.org/citation.cfm?id=2999134.2999257`.

Siddharth Krishna Kumar. On weight initialization in deep neural networks, 2017.

Ali Sharif Razavian, Hossein Azizpour, Josephine Sullivan, and Stefan Carlsson. Cnn features off-the-shelf: An astounding baseline for recognition. *2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2014. doi: 10.1109/cvprw.2014.131.

Lorenzo Rosasco, Ernesto De Vito, Andrea Caponnetto, Michele Piana, and Alessandro Verri. Are loss functions all the same? *Neural Computation*, 16(5): 1063–1076, 2004. doi: 10.1162/089976604773135104.

Raman Samusevich. Game theoretic optimization of detecting malicious behavior, 2016.

Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015. doi: 10.1109/cvpr.2015.7298682.

Sho Sonoda and Noboru Murata. Neural network with unbounded activation functions is universal approximator, 2015.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, January 2014. ISSN 1532-4435. URL `http://dl.acm.org/citation.cfm?id=2627435.2670313`.

Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, (9), 2008.

Peter N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1993.

7. Appendix A: Implementation of experiments

Programs

The CD attached to this thesis contains two sample programs:
classification.py (Attachment 1) implementing the training and evaluation of the classification experiments and
similarity.py (Attachment 2) implementing the class of neural networks mentioned in the similarity search chapter.

We also provide sample trained networks (Attachment 3) for each of the programs and a small proportion of training and testing dataset containing the GMM600 data (Attachment 4)

Both programs require Python 2.7 and TensorFlow 1.2. It is likely that they will also work with different versions, although we cannot guarantee that especially because of the rapid development of TensorFlow.

Control of classification.py

The program classify.py (Attachment 1) can train a neural network to classify the data or evaluate the performance of such a network. It is controlled from command line.

The first argument has value either "train" or "test", and specifies whether to train or evaluate the network. Each of the modes accepts different arguments.

Training

Creates and trains a new network with specified shape. The accepted argument are:

--model name of file where to store the trained neural network.
--shape shape of the network - number of layers, their activation functions and numbers of neurons. It is of form described by regular expression
$([1-9][0-9]^*X(\_sigmoid|\_relu)?(\_dropout)?X)^*$
$([1-9][0-9]^*X(\_sigmoid|\_relu)?(\_dropout)?)$. The default for each layer is ReLU and no dropout.
--batch_size number of samples in each batch
--epoch_duration duration of period between decreases of the learning rate (default 200,000)
--number_of_epochs number of epochs before stopping the training (default 8)
--learning_rate learning rate (default 0.0005)
--clean_data clean data used for training
--malware_data malicious data used for training

Example:
python classify.py train --model tfsaves/network1.tfsave

--shape 600_dropout_reluX1_sigmoid --epoch_duration 200000
--number_of_epochs 8 --clean_data dataset/gmm600_clean_train.csv
--malware_data dataset/gmm600_malware_train.csv
This command trains a network 600 Dropout Relu X 1 sigmoid for 8 epochs of duration 200,000 on the given data and stores it into tfsaves/network1.tfsave.

Evaluation

Evaluates an existing network. The accepted argument are:
--model, --shape have the same meaning
--clean_data clean data used for evaluation
--malware_data malicious data used for evaluation

Example:
python classify.py test --model tfsaves/network1.tfsave
--shape 600_dropout_reluX1_sigmoid --clean_data dataset/gmm600_clean_test.csv
--malware_data dataset/gmm600_malware_test.csv

Evaluates a network with shape 600 Dropout Relu X 1 sigmoid on the given dataset.

Control of similarity.py

The program similarity.py (Attachment 2) can train a neural network for similarity search or evaluate the performance of such a network. It is controlled from command line.

The first argument has value either "train" or "test", and specifies whether to train or evaluate the network. Each of the modes accepts different arguments.

Training

Creates and trains a new network with specified shape. The accepted argument are:
--model, --shape, --number_of_epochs have, --batch_size, --epoch_duration,
--learning_rate, -- clean_data, -- malware_data have the same meaning as in training of classifiers.
--selected_layer number of layer from which we should take the output (the first layer has number 1. 0 means input of the network)
--alpha α as described in Chapter 5.
--size_of_set number of randomly chosen malicious samples from which to choose the m_2 (see the chapter 5 for details). Default is 0, which disables this function.

Example:
python similarity.py train --model tfsaves/network2.tfsave
--shape 600_sigmoidX400_sigmoidX600_sigmoidX600_sigmoid
--epoch_duration 200000 --number_of_epochs 8

--clean_data dataset/gmm600_clean_train.csv
--malware_data dataset/gmm600_malware_train.csv --selected_layer 2 --alpha 0.4

Trains a siamese network 600 Sigmoid X 400 Sigmoid [OUT] X 600 Sigmoid X 600 sigmoid with $\alpha = 0.4$ for 8 epochs of duration 200,000 on the given data and stores it into tfsaves/network2.tfsave.

Evaluation

Evaluates an existing network. The accepted argument are:
--model, --shape, --clean_data, --malware_data, --selected_layer have the same meaning as in evaluation of classifiers.

Example:
python similarity.py test --model tfsaves/network2.tfsave
--shape 600_sigmoidX400_sigmoidX600_sigmoidX600_sigmoid
--clean_data dataset/gmm600_clean_test.csv
--malware_data dataset/gmm600_malware_test.csv --selected_layer 2

Evaluates a siamese network 600 Sigmoid X 400 Sigmoid [OUT] X 600 Sigmoid X 600 sigmoid on a given dataset.