Convolutional Neural Networks for Malware Classification

Daniel Gibert

Director: Javier Bejar
Department of Computer Science

A thesis presented for the degree of Master in Artificial Intelligence

Facultat d'Informàtica de Barcelona (FIB)
Facultat de Matemàtiques (UB)
Escola Tècnica Superior d'Enginyeria (URV)

Escola Politècnica de Catalunya (UPC) - BarcelonaTech
Universitat de Barcelona (UB)
Universitat Rovira i Virgili (URV)

20 October 2016

Abstract

According to AV vendors malicious software has been growing exponentially last years. One of the main reasons for these high volumes is that in order to evade detection, malware authors started using polymorphic and metamorphic techniques. As a result, traditional signature-based approaches to detect malware are being insufficient against new malware and the categorization of malware samples had become essential to know the basis of the behavior of malware and to fight back cybercriminals.

During the last decade, solutions that fight against malicious software had begun using machine learning approaches. Unfortunately, there are few open-source datasets available for the academic community. One of the biggest datasets available was released last year in a competition hosted on Kaggle with data provided by Microsoft for the Big Data Innovators Gathering (BIG 2015). This thesis presents two novel and scalable approaches using Convolutional Neural Networks (CNNs) to assign malware to its corresponding family. On one hand, the first approach makes use of CNNs to learn a feature hierarchy to discriminate among samples of malware represented as gray-scale images. On the other hand, the second approach uses the CNN architecture introduced by Yoon Kim [12] to classify malware samples according their x86 instructions. The proposed methods achieved an improvement of 93.86% and 98,56% with respect to the equal probability benchmark.

Acknowledgments

I would first like to thank my family, especially Mom, for the continuous support she has given me throughout my time in graduate school. Second, I would like to express my gratitude to my supervisor, Dr. Javier Béjar for their guidance during the course of this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Malware, short for malicious software, refers to software programs designed to damage or do any kind of unwanted actions on a computer system such as disrupting computer operations, gather sensitive information, bypass access controls, gain access to private computer systems and display unwanted advertising. Malware can be divided into the following categories not mutually exclusive depending on their purpose.

- Adware. It is a type of malware that automatically delivers advertisements. Advertising-supported software often comes bundled with software and applications and most of them serve as a revenue tool.
- Spyware. It is a type of malware that spies and track user activity without their knowledge. The capabilities of spyware can include keystrokes collection, financial data harvesting or activity monitoring.
- Virus. A virus is a type of malicious software capable of copying itself and spreading to other computers. Viruses can spread through email attachments, through the network the computer is connected if any other computer inside the network has been infected, by downloading software from malicious sites, etc.

- Worm. It is a type of malware that they spread through the computer network by exploiting operation system vulnerabilties. The major difference between worms and viruses is that computer worms have the ability to self-replicate and spread independently while viruses rely on human activity to spread.

- Trojan. A Trojan horse is a type of malware that disguises itself as a normal file or program to trick users into downloading and installing malware. A trojan can give unauthorized access to the infected computer and be used to steal data (logins, financial data, even electronic money), install more malware, modify files, monitor user activity (screen watching, keylogging, etc), use the computer in botnets, and anonymize internet activity by the attacker.

- Rootkit. A rootkit is a type of malicious software designed to remotely access or control a computer without being detected by users or security programs. For example, Rootkits can prevent a malicious process from being visible or it can keep its files from being read, etc.

- Backdoors. A backdoor is a computer software designed to bypass normal authentication procedures and compromise the system. Once a system has been compromised, one or more backdoors may be installed in order to allow access in the future without being detected by the user.

- Ransomware. It is a type of malicious software that essentially restricts user access to the computer by encrypting the files or locking down the system while demanding a ransom. Users are forced to pay the malware author to remove the restrictions and gain access to their computer. This type of payment is usually done with Bitcoins.

- Command & Control Bot. Bots are software programs created to automatically perform specific operations. Bots are commonly used for DDoS attacks, spambots that render advertisements on websites, as web spiders or for distributing malware. One way to defend against bots is by using CAPTCHA tests in websites to verify users as human.

Nowadays, the detection of malicious software is done mainly with heuristic and signature-based methods that struggle to keep up with malware evolution. Signature-based methods haven been heavily used for antivirus software for decades. A malware signature is an algorithm or hash that uniquely identifies a specific virus. While identifying a particular virus is advantageous it is quicker to detect a virus family through a generic signature. Virus researchers found that all viruses in a family share common behaviors and a single generic signature can be created for them. However, malware authors always try to stay a step ahead of AV software by writing polymorphic and metamorphic malware to do not match virus signatures. On one hand, polymorphic code uses a polymorphic engine to mutate while keeping the original algorithm intact. Encryption is the most common way to hide code. On the other hand, metamorphic viruses translate their own binary code into a temporary representation, editing the temporary representation of themselves and then translate the edited form back to machine code again. This mutation can be done by using techniques such as changing what registers to use, changing machine instructions to equivalent ones, inserting NOP instructions or reordering independent instructions.

Therefore, AV vendors rely also in heuristic-based methods. This approach is based on rules determined by experts which rely on dynamic analysis of malicious behavior and thus, it can deal with unknown malware but it also generates greater amounts of false positives than signature-based methods because not each one of the detected suspicious executable files is a malware file. As a result, AV vendors attempted to use an hybrid analysis approach by using both signature-based and heuristic-based methods to tackle with unknown malware. In contrast, before creating the signatures for malware, it has first to be analyzed so to understand its capabilities and behavior. The program capabilities and behavior can be observed either by examining its code or by executing it in a safe environment.

1. Static Analysis. It refers to the analysis of a program without executing it. The patterns detected in this kind of analysis include string signature, byte-sequence or opcodes frequency distribution, byte-sequence n-grams or opcodes n-grams, API calls, structure of the disassembled program, etc. The malicious program is usually unpacked and decripted before doing static analysis by using disassembler or debugger tools such as IDA Pro or OllyDbg which can be used to reverse compiled Windows executables and display malware code as a sequence of Intel x86 assembly instructions.

2. Dynamic Analysis. It refers to the analysis of the behavior of a malicious program while it is being executed in a controlled environment (virtual machine, emulator, sandbox, etc). The behavior is monitored by using tools like Process Monitor, Process Explorer, Wireshark or Capture BAT. This kind of analysis tries to monitor function and API calls, the network, the flow of information, etc. Compared to static analysis, it is more effective and does not require the executable to be disassembled but on the other hand, it takes more time and consumes more resources than static analysis, being more difficult to scale. In addition, as the controlled environment in which the malware is monitored is different from the real one the program may behave different. That's because some behavior of malware might be triggered only under certain conditions such as via a specific command or on a specific system date and in consequence, can't be detected in a virtual environment

In recent years, the possibility of success of machine learning approaches has increased thanks to the confluence of three developments: (1) the rise of commercial feeds that provide volumes of new malware, (2) the computing power has become cheaper meaning that researchers can fit large and more complex models to data and (3) machine learning as discipline has evolved and there are more tools at their disposal. Machine learning approaches hold the promise that they might achieve high detection rates without the need of human signature generation required by traditional approaches. In consequence, AV companies and researchers begun to employ machine learning classifiers to help them address this problem such as logistic regression[22], neural networks[8] and decision trees[14].

The two principal tasks that have been carried out within the scope of malware analysis are (1) malware detection and (2) malware classification. First, a file needs to be analyzed to detect if has any malicious content. In case it exhibits any malicious content it is assigned to the most appropriate malware family according to their content and behavior through a classification mechanism.

1.1 Objective

This master thesis aims to explore the problem of malware classification. In particular, this thesis proposes two novel approaches based on Convolutional Neural Networks (CNNs). On one hand, CNNs were applied for learning discriminative patterns from malware images based on the work performed by Nataraj et al.[21]. On the other hand, the CNN architecture proposed in [12] was used to classify malicious software based on malware's x86 instructions. Both approaches have been evaluated on the data provided by Microsoft for the BIG Cup 2015 (Big Data Innovators Gathering).

1.2 Organization

The thesis is organized following chapters. The first and current chapter is the introduction, which also contains the objectives and the organization of the thesis. The second chapter introduces the background of the project, focusing on neural networks and deep learning from its beginning until now. The third chapter presents the state of the art review with special attention on the machine learning algorithms and features used to detect and classify malware. The fourth chapter introduces the Kaggle platform and the Microsoft's Malware Classification Challenge. In addition, it also describes two solutions of the competition. The fifth chapter describes the approach based on the representation of malware as gray-scale images and the sixth chapter explains how Convolutional Neural Networks can be used to extract features from malware's x86 instructions represented via word embeddings. Finally, the last chapter wraps up the conclusions and the future work to be done.

Chapter 2

Background

Nowadays Deep Learning is the hottest topic in the Artificial Intelligence and Machine Learning field. Deep Learning refers to the set of techniques used for learning in Neural Networks with many layers. However, it is based on a set of previous ideas that appeared over the 60s. In this chapter are explained the most important concepts behind deep learning.

2.1 Artificial Neural Networks

Artificial Neural Networks are a family of models inspired by the way biological nervous systems, such as the brain, process information which enables a computer to learn from data. There are different types of NNs but in this thesis are only presented two of them: (1) feed-forward networks and (2) convolutional neural networks. First it is introduced the architecture of a feed-forward network. Thus, this type of nets are composed by at least three layers, one input layer, one output layer and one or more hidden layers. A feed-forward network has the following characteristics:

- Neurons are arranged in layers, with the first layer taking in inputs and the last layer producing the output.

- Each neuron in one layer is connected to every neurons in the next layer.

- There is no connection between neurons in the same layer.

To understand how a neural network works, first it has to be understood what are neurons, how neurons learn and how the information pass from one neuron to another.

2.1.1 Perceptrons

A perceptron was the earliest supervised learning algorithm and it is the basic building block of Artificial Neural Networks (ANN). It was first introduced in 1957 at the Cornell Aeronautical Laboratory by Frank Rosenblatt.

It works by taking several inputs $(x_1, x_2, ..., x_j)$ and producing a single output (y). Rosenblat introduced weights $(w_1, w_2, ..., w_j)$ to express the importance of the respective inputs to the output. The output of the perceptron is either 0 or 1 and it is determined by whether the weighted sum $\sum_j w_j * x_j + b$ is less than or greater than 0.

$$output = \begin{Bmatrix} 1: & if \ w * x + b > 0 \\ \\ 0: & if \ w * x + b <= 0 \end{Bmatrix}$$

However, a perceptron can only output zero or one, making impossible to extend the model to work on classification tasks with multiple categories. This issue can be solved by having multiple perceptrons in a layer, such that all these perceptrons receive the same input and each one is responsible for one output of the function. In fact, Artificial Neural Networks (ANNs) are nothing more than layers of perceptrons (neurons or units as called nowadays). A perceptron can be seen as an ANN with only one layer, the output layer, with 1 neuron.

2.1.2 Sigmoid neuron

The main limitation of perceptrons is that there are very difficult to tune, because minimum changes in the weights and bias of any single perceptron can cause the output to change drastically by completely flip, from 0 to 1 or viceversa. And if we have a network of perceptrons, a single flip can completely change the behavior of the rest of the network.
This problem was solved by the introduction of the sigmoid neuron.

Exactly as the perceptron, a sigmoid neuron has inputs $(x_1, x_2, ..., x_j)$ and it also has weights for each input and a bias, but the output can be a real number. The sigmoid function is defined as:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

And the output of the sigmoid neuron is:

$$\frac{1}{1 + exp(-\sum_j w_j * x_j - b)}$$

Hence, the only difference between the perceptron and the sigmoid neuron is the activation function.

2.1.3 Loss function

To measure the performance of the neural network it is defined a function, typically named cost or loss function which given a prediction or set of predictions and a label or a set of labels measures the discrepancy between the algorithms prediction and the correct label. There are various cost functions but the most common and simple in neural networks is the mean squared error (MSE).

The mean squared error can be defined as:

$$L(W,b) = 1/m(\sum_{i=1}^{m} ||h(x^i) - y^i||^2)$$

where:

- m is the number of training examples
- x^i is the i^{th} training sample
- y^i is the class label for the i^{th} training sample
- $h(x^i)$ is the algorithm's prediction for the i^{th} training sample

The goal in training neural networks is to find weights and biases that minimizes some cost/loss function C. For that, it is used an algorithm called gradient descent.

2.1.4 Gradient Descent Algorithm

Gradient Descent is an algorithm for minimizing the loss function. It is used to find the local minimum of the loss function. Next you will find an outline of the algorithm.

1. Start with a random initialization of each weight and bias in the NN. It is important to randomly initialize all parameters because if not, if all parameters start off at identical values, then all the hidden layer units will end up learning the same function of the input. In consequence, random initialization serves the purpose of symmetry breaking.

2. Keep iterating to update the parameters W,b as follows until it hopefully ends up at a minimum:

$$W_{i,j}^{l} = W_{i,j}^{l} - \alpha \frac{\partial}{\partial W_{i,j}^{l}} L(W,b)$$

$$b_i^l = b_i^l - \alpha \frac{\partial}{\partial b_i^l} L(W, b)$$

where α is the learning rate and $W_{i,j}^l$ and b_i^l denote each weight and bias in a particular layer l in the NN, respectively.

The derivative of the overall loss function can be computed as:

$$\frac{\partial}{\partial W_{i,j}^l} L(W, b) = [\frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial W_{i,j}^l} L(W, b : x^i, y^i)]$$

$$\frac{\partial}{\partial b_i^l} L(W, b) = [\frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial b_i^l} L(W, b : x^i, y^i)]$$

The learning rate is used to control how big a step is taken downhill with gradient descent. Selecting the correct learning rate is critical. On one hand, if α is too small, gradient descent can be slow. On the other hand, if α is too large, gradient descent can overstep the minimum and even diverge.

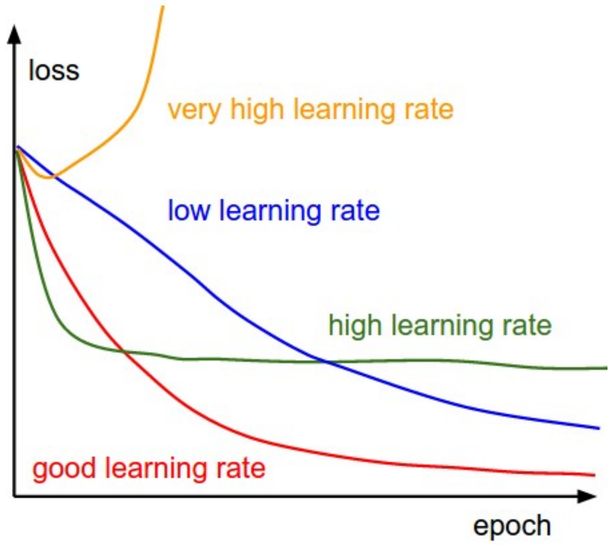


Figure 2.1: Effects of different learning rates

2.1.5 Backpropagation

The key step is to compute all those partial derivates presented before. Therefore, to compute efficiently these partial derivates is used the backpropagation algorithm.

The intuition behind is as follows. Given a training example (x^i, y^i) first of all it is ran a forward pass to compute all the activations through the network (also the output value of the hypothesis $h(x^i)$). Thus, for each node i in layer l it is computed an error term, δ_i^l, that measures how much that node was responsible for any errors in the output. For an output node, the error term is measured directly from the difference between the network's activation and the true target value and is defined as $\delta_i^{n_l}$, where n_l is the output layer. For hidden units, it is computed δ_i^l based on the weighted average of the error terms of the nodes that use a_i^l as an input (a_i^l = activation of node i in layer l).

Overview of the algorithm:

1. Compute the activation of the layers l_1, l_2 and so on up to the output layer l_{nl} by performing a feedforward pass.

2. For each output unit i in layer nl compute the error term δ_i^{nl}.

$$\delta_i^{nl} = \frac{\partial}{\partial z_i^{n_l}} * \frac{1}{2}||y - h_{W,b}(x)||^2$$

3. For all hidden layers $l = n_l - 1, n_l - 2, ..., 2$. Compute for each node i in layer l:

$$\delta_i^l = \left(\sum_{j=1}^{l+1} W_{ji}^l \delta_j^{l+1} \right) * f^{'}(z_i^l)$$

4. Finally, compute the desired derivatives given as:

$$\frac{\partial}{\partial W_{i,j}^{l}} L(W, b : x, y) = a_j^l * \delta_i^{l+1}$$

$$\frac{\partial}{\partial b_i^l} L(W, b : x, y) = \delta_i^{l+1}$$

2.2 Convolutional Neural Networks

A Convolutional Neural Network (CNN) is a type of feed-forward NN in which the connectivity pattern between its neurons is inspired by the organization of the animal visual cortex, whose individual neurons are arranged in such a way that they respond to overlapping regions tilling the visual field.

CNN are composed by three types of layers: (1) fully-connected, (2) convolutional (3) and pooling. All the various implementations of CNN can be loosely described as involving the following process:

1. Convolve several small filters on the input image.
2. Subsample this space of filter activations.
3. Repeat steps 1 and 2 until you are left with enough high level features.
4. Apply a standard feed-forward NN to the resulting features.

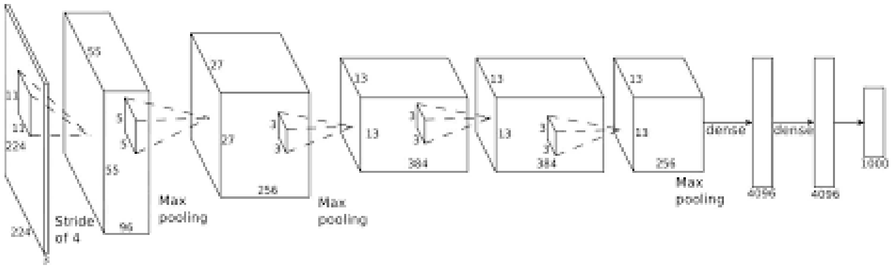


Figure 2.2: AlexNet architecture

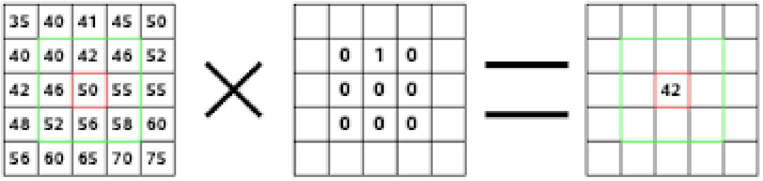
Figure 2.2 corresponds to the architecture used in [16] that was applied to the ImageNet classification contest. The architecture consists of 8 learnable layers, the first five are convolutional and the rest are fully-connected layers.

2.2.1 Local connectivity

It is impractical to connect neurons to all neurons in the previous layer when dealing with high-dimensional inputs like images because in such network architectures the spatial structure of the data is not taken into account.
In convolutional neural networks each neuron is connected to a small region of the input neurons (each neuron connects only to a small contiguous region of pixels in the input) and thus, CNNs are able to exploit spatially local correlation by enforcing a local connectivity pattern between neurons of adjacent layers.

2.2.2 Convolutional Layer

Convolutional layers are the core of a CNN. A convolutional layer consists of a set of learnable kernels which are convolved across the width and the height of the input features during the forward pass producing a 2-dimensional activation map of the kernel.
As a summary, a kernel consists of a layer of connection weights with the input being the size of a small 2D patch and the output being a single unit.



35	40	41	45	50
40	40	42	46	52
42	46	50	55	55
48	52	56	58	60
56	60	65	70	75

	0	1	0	
	0	0	0	
	0	0	0	

		42		

Figure 2.3: Convolution

In figure 2.3 it is represented how convolution works. Considering an image of 5x5 pixels as in the figure, where 0 means values completely black and 255 means completely white. In the center of the figure, it has been defined a kernel of 3x3 pixels with all eight 0 except one wright set at 1. The output is the result of computing the kernel at each possible position in the image.

Whether or not the kernel is convolved through all positions it is determined by the stride. For example, for stride 1, it outputs the typical convolution but for stride 2 half of the convolutions are avoided because there should be 2 pixels of distance between centers.

The size of the output after convolving a kernel of size Z over an image N with stride S is defined as:

$$output = \frac{N - Z}{S} + 1$$

2.2.3 Pooling Layer

Pooling is a form of non-linear down-sampling. There are several non-linear functions to implement pooling such as the minimum, the maximum and the average but the most common is the maximum. Max pooling works by partitioning the image into a set of non-overlapping rectangles and for each sub-region outputs the maximum value.

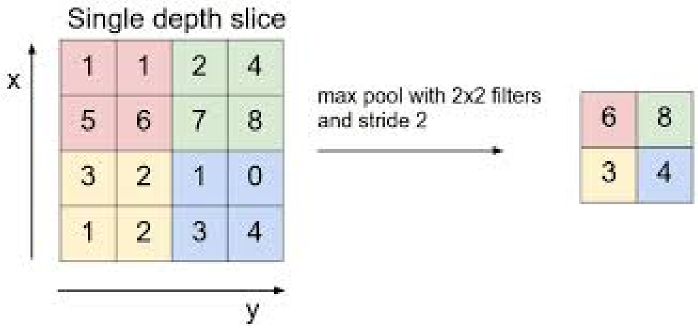


Figure 2.4: Max pooling

The main benefits of max-pooling are:

1. It reduces computation for upper layers by eliminating non-maximal values.

2. It provides a form of translation invariance and in consequence, provides additional robustness to position being a way of reducing the dimensionality of intermediate representations.

2.3 Overfitting

Overfitting refers to the condition a predictive model describes the random noise of a particular data instead of learning the underlying relationship. As a result, these models may not yield accurate predictions for new observations.

This section describes the most common techniques used to avoid overfitting in large networks.

2.3.1 Regularization

Regularization adds an extra term, named regularization term to the loss function in a way that in consequence, the network would prefer to learn small weights and penalize large weights. Regularization usually doesn't affect biases. That's because large biases make it easier for neurons to saturate, which is sometimes desirable. Moreover, having large biases doesn't make a neuron sensitive to its inputs in the same way as having large weights.

- L2 regularization.

$$L(W, b) = L(W, b)_0 + \frac{\lambda}{2n} \sum_w w^2$$

- L1 regularization.

$$L(W, b) = L(W, b)_0 + \frac{\lambda}{n} \sum_w |w|$$

where $L(W, b)_0$ is the original unregularized loss function.

2.3.2 Dropout

In large neural networks, it is difficult to average the predictions of different networks at test time. To address this problem, dropout was introduced in [32] by Geoffrey Hinton. The idea behind is to randomly drop units (along with their connections) from the neural network during training to prevent neurons from co-adapting too much.

At each iteration, we randomly and temporaly disconnect a percentage of the hidden neurons (usually, half of them) in the network except those in the input and the output layer. After that, the input is propagated through the modified network and then the result is backpropagated also through the modified network. After updating the appropiate weights and biases, the process is repeated by, first restoring the dropout neurons, then choosing a new random subset of hidden neurons to disconnect and so on.

Heuristically, the result of dropout is like training different neural networks and in consequence, the dropout procedure is like averaging the effects of a large number of networks. These networks will overfit in different ways but at the end, hopefully, the network effect of dropout will be to reduce overfitting. Lastly, by reducing neurons co-adaptations, a neuron cannot rely on the presence of a particular other neuron and is forced to learn more robust features that are useful in conjunction with the other different random subsets of neurons.

2.3.3 Artificially expanding the training data

One of the best ways of reducing overfitting is to increase the size of the training data. Unfortunately, training data sometimes is expensive or difficult to acquire. To attach this problem, training data can be artificially expanded. In particular, in recent competitions such as CIFAR-10, training images were cropped, flipped and transposed in various ways to expand the training data.

2.4 Deep Learning

As defined at the beginning of the chapter, deep learning is the term used to describe the techniques used to learn in networks with many layers (deep neural networks). What convert a neural network into a deep neural network is basically the number of layers. However, if deep neural networks are just a NN with more layers, why it has been until recently that they have attracted that much attention? Well, it is mainly because of two reasons: (1) the computational power needed to train this networks and (2) the vanishing gradient problem.

1. The basic idea of software able to simulate the neocortex's neurons in an artificial neural network is decades old but because of the improvements in mathematical formulas and the increasingly powerful computers, computer scientist have been able to model networks with many more layers than before.

2. The vanishing gradient problem was introduced in [10] by Sepp Hochreiter which found that in a neural network with activation functions such as the sigmoid or the hyperbolic tangent where the gradient range is $(-1, 1)$ or $[0, 1)$, backpropagation is computed by the chain rule, multiplying n of this small numbers from the output layer through a n-layer network, meaning that gradient decreased exponentially with n. This results in the front layers training much more slowly than other layers.

2.4.1 ReLU units

The vanishing gradient problem was not solved until 2010 by the introduction of the Rectified Linear units (ReLU) [9]. The activation function of ReLU units is defined as:

$$f(x) = \sum_{i=1}^{\infty} \sigma(x - i + 0.5) \approx log(1 + e^{x})$$

The softmax function $log(1 + e^x)$ can be approximated by the max function or hard-max function $f(x) = max(0, x)$.

Figure 2.5 presents a graphically comparison between the ReL and the sigmoid function.

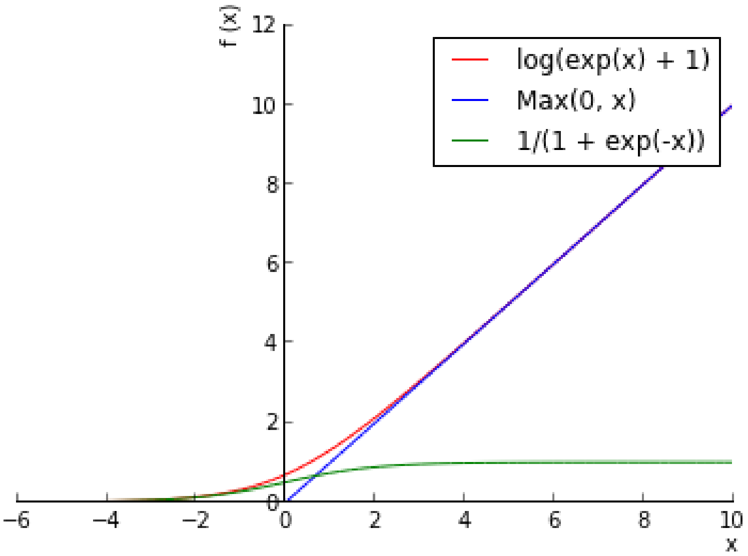


Figure 2.5: ReL and sigmoid functions comparison

The main difference between both functions is that:

- The sigmoid function has range $[0, 1]$ while the ReL function has range $[0, \infty)$.
- The gradient of the sigmoid function vanishes as it is increased or decreased x while the gradient of the ReL function doesn't vanish as x is increased.

2.4.2 Gradient Descent Optimization Algorithms

There are three variants of the gradient descent algorithm, which differ in how much data is used to compute the gradient of the objective function.

1. Batch Gradient Descent. It computes the gradients for the loss function L(W,b) for the entire training set. It guarantees the convergence to the global minimum for convex error surfaces and to a local minimum for non-convex surfaces.

2. Stochastic Gradient Descent. It performs a parameter update for each training example x^i and label y^i. It is used for online learning as it performs one update at a time. It enables to jump to new and potential better local minimum but complicates the convergence to the exact local minimum.

3. Mini-batch Gradient Descent[17]. It computes the gradients for the loss function L(W,b) only for a small batch of n training samples. Its faster than batch gradient descent and leads to a better convergence than stochastic gradient descent.

Next it is presented an outline of some algorithms used in deep learning to optimize the gradient descent algorithm.

1. Momentum [25]

   The simplest gradient algorithm known as steepest descent 2.1.4, modifies the weight at time step t according to:

   $$W^l_{i,j} = W^l_{i,j} - \alpha \frac{\partial}{\partial W^l_{i,j}} L(W, b)$$

   $$b^l_i = b^l_i - \alpha \frac{\partial}{\partial b^l_i} L(W, b)$$

   However, it is known that learning such scheme can be very slow. To improve the speed of convergence of the gradient descent algorithm it is included the momentum term in the formula:

   $$W^l_{i,j,t+1} = W^l_{i,j,t} - \alpha \frac{\partial}{\partial W^l_{i,j}} L(W, b) + \gamma W^l_{i,j,t-1}$$

$$b^l_{i,t+1} = b^l_{i,t} - \alpha \frac{\partial}{\partial b^l_i} L(W, b) + \gamma b^l_{i,t}$$

where γ is the momentum term. In consequence, the modification of the weight vector at the step t depends on both the current gradient and the weight change of the step $t-1$.

2. Adagrad [5]

   Adagrad is an algorithm for gradient-based optimization that adapts the learning rate to the parameters, performing smaller updates for frequent parameters and larger updates for infrequent parameters. Adagrad uses a different learning rate for every parameter $W^l_{i,j,t}$ at each time step t. In its update rule, it modifies the general learning rate α at each time step t for every parameter $W^l_{i,j,t}$ based on the past gradients that have been computed for $W^l_{i,j,t}$.

$$W^l_{i,j,t+1} = W^l_{i,j,t} - \frac{\alpha}{G^l_{t,ij} + \epsilon} * \frac{\partial}{\partial W^l_{i,j}} L(W, b)$$

   where $G^l_{t,ij} \in \mathbb{R}^{dxd}$ is the diagonal matrix where each diagonal element ij is the sum of the squares of the gradients $W^l_{i,j,t+1}$ up to time step t 24 and ϵ is smoothing term that avoids division by zero ($\approx 1e-8$).

3. Adam [13]

   Adam is the acronym for Adaptive Moment Estimation. It is another method that computes adaptive learning rates for each parameter. It stores an exponentially decaying average of the past squared gradients that we will denota v_t and similar to momentum, it keeps an exponentially decaying average of past gradients m_t:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) * g_t v_t = \beta_2 v_{t-1} + (1 - \beta_2) * g_t^2$$

   and g_t is the gradient of the objective function and β_1 and β_2 are the decay rates. m_t and v_t are the estimates of the first moment or mean and the second moment or uncentered variance of the gradients, respectively. m_t and v_t are initialized as vectors of 0's and in consequence, during initial time steps and when the decay rates are small they tend to be biased towards 0. To solve this problem, they computed the bias-corrected first and second moment estimates:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

As a result, the Adam update rule is defined as:

$$W_{t+1} = W_t - \frac{\alpha}{\sqrt{\hat{v}_t + \epsilon}} * \hat{m}_t$$

Chapter 3

State of the Art

During the last decade, researchers and anti-virus vendors have begun employing machine learning algorithms like the Association Rule, Support Vector Machines, Random Forests, Naive Bayes and Neural Networks to address the problem of malicious software detection and classification. An overview of the methods can be found in [26], [7] and [6]. Following a few of these approaches used in literature are discussed.

1. Byte-sequence N-grams[33, 31]
   The representation of a malware file as a sequence of hex values can be effectively described through n-gram analysis. A N-gram is a contiguous sequence of n hexadecimal values from a given malware file. Each element in the byte sequence can take one out of 257 different values, i.e. the 256 byte range plus the special '??' symbol. Byte-sequence N-grams were first presented in [33], where they proposed a N-gram based algorithm for malware classification implemented in IBM's antivirus scanner. The algorithm used 3-grams as features and a neural network as a classification model. Notice that the size of the features increase exponentially being 256^2 for a bigram model and 256^3 for a trigram model being the techniques for dimensionality reduction a must in this kind of models.

2. Opcodes N-grams [11, 27, 2, 3, 30]

   Similar to byte-sequence N-grams, n-gram models have been generated from opcodes extracted from assembly language code files. An opcode (abbreviated from operation code) is the portion of a machine language instruction that specifies the operation to be performed. In particular, [3] investigated the most frequent opcodes and the rare opcodes present in both goodware and malware. The following two charts show the 14 most frequent opcodes in goodware and in malware, respectively.

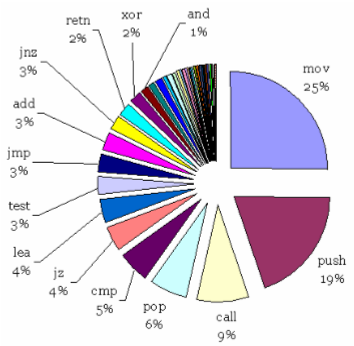


Figure 3.1: Most frequent 14 opcodes for goodware

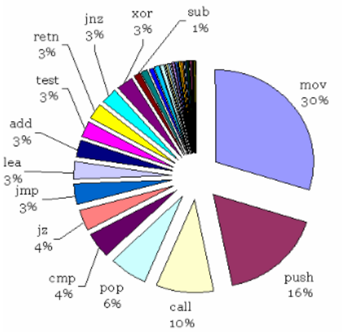


Figure 3.2: Most frequent 14 opcodes for malware

3. Portable Executable [37, 28]

   PE format is a file format for executables, DLLs and object code commonly used in the Windows operating systems. PE format is a data format that encapsulates the necessary information for the Windows OS loader to manage the executable code. It includes information such as dynamic library references for linking and API import and export tables.

Features from Portable Executables (PE) are extracted by performing static analysis using structural information of PE and are useful to indicate whether or not the file has been manipulated or infected to perform malicious activity. In [37] they extracted the following features from PE: (1) File pointers which denote the position within the file as it is stored on disk; (2) Import sections which describe functions from which DLLs were used and the list of DLLs of the executable that are imported; (3) Exports section which describes the functions that are exported; (4) Structure of the PE header such as features like the code and file size, the creation time, etc. After the feature extraction process they build an ensemble of support vector machines. The training set consisted of 9838 executables, from which 2320 were benign executables, 1936 backdoors, 1769 spyware, 1911 trojans and 1902 worms. They tested the performance of the classifier on two datasets: (1)Malfease and (2)VXheavens.

In [28], Invencea Labs build a Deep Neural Network to detect malware by using a set of features derived from the numerical fields extracted from the file binary's PE packaging. The Deep NN consisted of three layers: the input layer of 1024 input features and two hidden layers of 1024 PReLU units each one. The output layer consisted of one sigmoid unit denoting whether the file is goodware or malware. In the following picture you can see an outline of the framework.

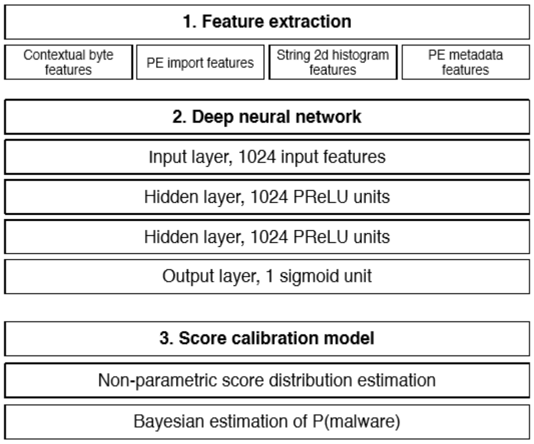

1. Feature extraction
Contextual byte features
PE import features
String 2d histogram features
PE metadata features
2. Deep neural network
Input layer, 1024 input features
Hidden layer, 1024 PReLU units
Hidden layer, 1024 PReLU units
Output layer, 1 sigmoid unit
3. Score calibration model
Non-parametric score distribution estimation
Bayesian estimation of P(malware)

Figure 3.3: Outline of Invencea's Malware Detection Framework

4. Entropy [29, 18]

   Malware authors use obfuscation techniques to pass through signature-based detection systems of antivirus programs. For this reason, it is examined the statistical variation in malware samples to identify packed and encrypted samples. In [18] it was presented a tool to analyze the entropy of each PE section in order to determine which executable sections might be encrypted or packed. They found that the average entropy is 4.347, 5.099, 6.801, 7.175 for plain files, native executables, packed executables and encrypted executables, respectively.

5. API calls [4, 36]

   API and function calls have been widely used to detect and classify malicious software. An experiment was conducted to determine the top maliciously used APIs. They retrieved the imports of all of the PE files and proceeded to count the number of times each sample uniquely imported an API. They found that there was a total of 120126 uniquely imported APIs.

   In [4] they proposed a malware detection system which uses Window

API call sequences. They used a 3rd order Markov chain, i.e. 4-grams, to model the API calls. The malicious executables mainly consisted of backdoors, worms and Trojan horses collected from VXHeavens. Their detection system achieved an accuracy of 90%.

6. Use of registers

   In [19], they proposed a method based on similarities of binaries behaviors. They assumed that the behavior of each binary can be represented by the values of memory contents in its run-time. In other words, values stored in different registers while malicious software is running can be a distinguishing factor to set it apart from those of benign programs. Then, the register values for each API call are extracted before and after API is invoked. After that, they traced the changes of registers values and created a vector for each of the values of EAX, EBX, EDX, EDI, ESI and EBP registers. Finally, by comparing old and unseen malware vectors they achieved an accuracy of 98% in unseen samples.

7. Call Graphs

   A call graph is a directed graph that represents the relationships between subroutines in a computer program. In particular, each node represents a procedure/function and each edge (f,g) indicates that procedure f call procedure g. This kind of analysis have been used for malware classification with good results. In [15], they presented a framework which builds a function call graph from the information extracted from disassembled malware programs. For every node (i.e. function) in the graph, they extracted attributes including library APIs calls and how many I/O read operations have been been made by the function. Then, they computed the similarity between any two malware instances.

8. Malware as an Image

   In [21] a completely different approach to characterize and analyze malicious software was presented. They represented a malware executable as a binary string of zeros and ones. Then, the vector was reshaped into a matrix and the malware file could be viewed as a gray-scale image. They were based on the observation that for many malware families, the images belonging to the same family appear to be very similar in layout and texture.

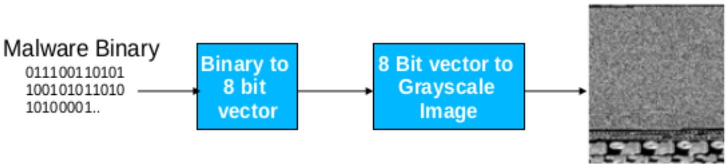


Figure 3.4: Visualizing Malware as an Image

   In addition, to compute texture features from malware images they used GIST[34, 23]. For classification, they used k-nearest neighbors with Euclidean distance and they obtained a classification rate of 0.9929 performing as state of the art results in the literature but at a significantly less computational cost.

Chapter 4

Microsoft Malware Classification Challenge

The content of this chapter is structured as follows. First of all, it is presented the Microsoft Malware Classification Challenge and the platform where the competition was hosted. Secondly, it is described the winner's solution of the challenge and the set of techniques they used to win the competition followed by an approach that achieved a logloss of 0.0064 and used features extracted from gray-scale images of malware. Lastly, it is presented the deep learning library used to implement the neural networks.

4.1 What's Kaggle?

Kaggle is a platform where a large community of data scientist comprised from thousands of MsCs and PhDs from fields such as computer science, statistics and maths compete to solve valuable problems. These problems come from competitions that companies host in Kaggle or from competitions that are part of class homework or projects in academic institutions.

4.2 Microsoft Malware Classification Challenge

In 2015, Microsoft hosted a competition in Kaggle with the goal of classifying malware into their respective families based on the their content and characteristics. For this challenge, Microsoft provided a dataset of 21741 samples, with 10868 for training and the other 10873 for testing, being a dataset of almost half a terabyte uncompressed. Microsoft provided a set of malware samples representing 9 different malware families. Each malware sample had an Id, a 20 character hash value uniquely identifying the sample and a class, an integer representing one of the 9 malware family names to which the malware belong: (1) Ramnit, (2) Lollipop, (3) Kelihos_ver3, (4) Vundo, (5) Simda, (6) Tracur, (7) Kelihos_ver1, (8) Obfuscator.ACY, (9) Gatak. The distribution of classes present in the training data is not uniform and the number of instances of some families significantly outnumbers the instances of other families.

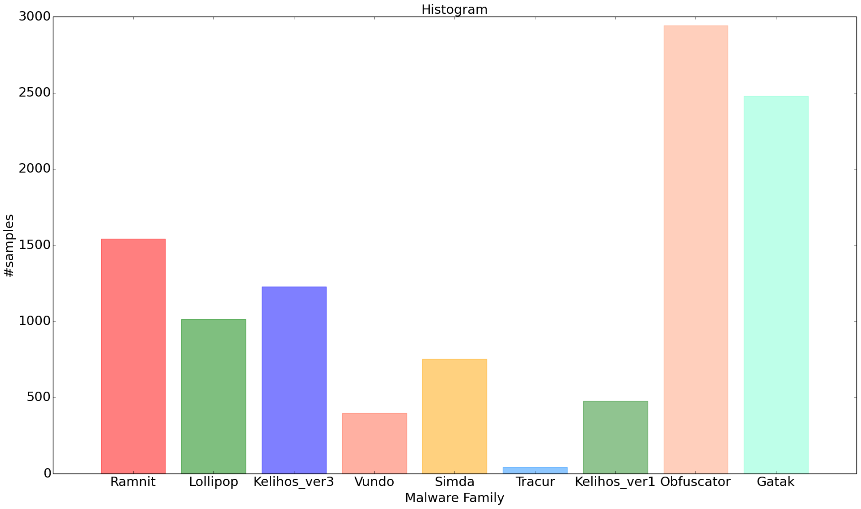
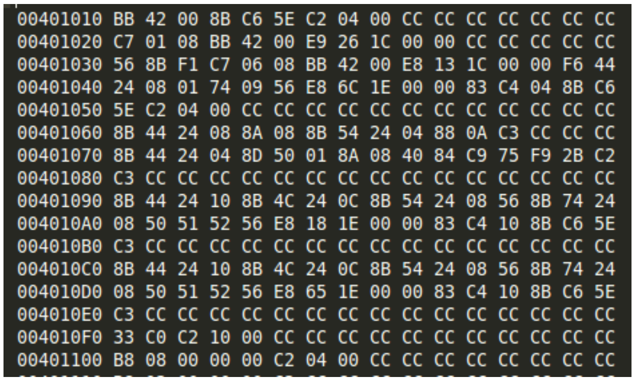


Figure 4.1: Malware Classification Challenge: dataset

For each observation we were provided with a file containing the hexadecimal representation of the file's binary content and a file containing metadata information extracted from the binary content, such as function calls, strings, sequence of instructions and registers used, etc, that was generated using the IDA disassembler tool.

4.2.1 Bytes file

The bytes file is the raw hexadecimal representation of the malware's binary content. A snapshot of one of these bytes files is shown below.



```
00401010 BB 42 00 8B C6 5E C2 04 00 CC CC CC CC CC CC CC
00401020 C7 01 08 BB 42 00 E9 26 1C 00 00 CC CC CC CC CC
00401030 56 8B F1 C7 06 08 BB 42 00 E8 13 1C 00 00 F6 44
00401040 24 08 01 74 09 56 E8 6C 1E 00 00 83 C4 04 8B C6
00401050 5E C2 04 00 CC CC CC CC CC CC CC CC CC CC CC CC
00401060 8B 44 24 08 8A 08 8B 54 24 04 88 0A C3 CC CC CC
00401070 8B 44 24 04 8D 50 01 8A 08 40 84 C9 75 F9 2B C2
00401080 C3 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC
00401090 8B 44 24 10 8B 4C 24 0C 8B 54 24 08 56 8B 74 24
004010A0 08 50 51 52 56 E8 18 1E 00 00 83 C4 10 8B C6 5E
004010B0 C3 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC
004010C0 8B 44 24 10 8B 4C 24 0C 8B 54 24 08 56 8B 74 24
004010D0 08 50 51 52 56 E8 65 1E 00 00 83 C4 10 8B C6 5E
004010E0 C3 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC
004010F0 33 C0 C2 10 00 CC CC CC CC CC CC CC CC CC CC CC
00401100 B8 08 00 00 00 C2 04 00 CC CC CC CC CC CC CC CC
```

Figure 4.2: Snapshot of one bytes file

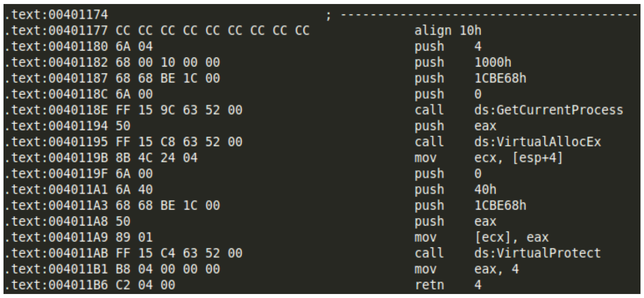
Each record of the hexadecimal files is composed by:

- Byte Count. Two hex digits indicating the number of hex digits pairs in the data field.
- Address. Four hex digits representing the 16-bit beginning memory address offset of the data.
- Record Type. Two hex digits, 00 to 05, defining the meaning of the data field.
- Data. A sequence of n bytes of data, represented by 2n hex digits.

- Checksum. Two hex digits, a computed value that can be used to verify the record has no errors.

4.2.2 ASM file

The asm file, generated by the disassembler tool, is a log containing various metadata such as rudimentary function calls, memory allocation, and variable manipulation. A snapshot of one of these files is shown below.



```
.text:00401174                              ; ----------------------------------------
.text:00401177 CC CC CC CC CC CC CC CC CC               align 10h
.text:00401180 6A 04                                    push    4
.text:00401182 68 00 10 00 00                           push    1000h
.text:00401187 68 68 BE 1C 00                           push    1CBE68h
.text:0040118C 6A 00                                    push    0
.text:0040118E FF 15 9C 63 52 00                        call    ds:GetCurrentProcess
.text:00401194 50                                       push    eax
.text:00401195 FF 15 C8 63 52 00                        call    ds:VirtualAllocEx
.text:0040119B 8B 4C 24 04                              mov     ecx, [esp+4]
.text:0040119F 6A 00                                    push    0
.text:004011A1 6A 40                                    push    40h
.text:004011A3 68 68 BE 1C 00                           push    1CBE68h
.text:004011A8 50                                       push    eax
.text:004011A9 89 01                                    mov     [ecx], eax
.text:004011AB FF 15 C4 63 52 00                        call    ds:VirtualProtect
.text:004011B1 B8 04 00 00 00                           mov     eax, 4
.text:004011B6 C2 04 00                                 retn    4
```

Figure 4.3: Snapshot of one assembly code file

An assembly program is often divided into three sections:

1. The data section. It is used to declare initialized data or constants and do not change at runtime.

2. The bss section. It is used for declaring variables. Contains uninitialized data.

3. The text section. It keeps the actual code of the program.

Apart from the previous sections, an assembly program can contain other sections such as:

- The rsrc section. It contains all the resources of the program.

- The rdata section. It holds the debug directory which stores the type, size and location of various types of debug information stored in the file.

- The idata section. It contains information about functions and data that the program imports from DLLs.
- The edata section. It contains the list of the funcions and data that the PE file exports for other programs.
- The reloc section. It holds a table of base relocations. A base relocation is an adjustment to an instruction or initialized variable value that's needed if the loader couldn't load the file where the linker assumed it would.

Additionally, some other sections can appear as a result of applying a polymorphic or metamorphic techniques to hide the actual code.

The assembly language consists of three types of statements:

1. Instructions or assembly language statements. Are used to tell the processor what to do. Instructions are entered one instruction per line. Each instruction has the following format:

   ```
   [label]    mnemonic    [operands]    [;comment]
   ```

   where the fields in brackets are optional. A basic instruction has two parts: (1) the name of the instruction or the mnemonic to be executed (also known as opcodes); (2) the operands or the parameters of the command.

   ```
   INC COUNT          ; Increment the memory variable COUNT
   MOV TOTAL, 48      ; Transfer the value 48 in the
                      ; memory variable TOTAL
   ```

   Next you will find the top 10 most used x86 instructions in the training dataset.

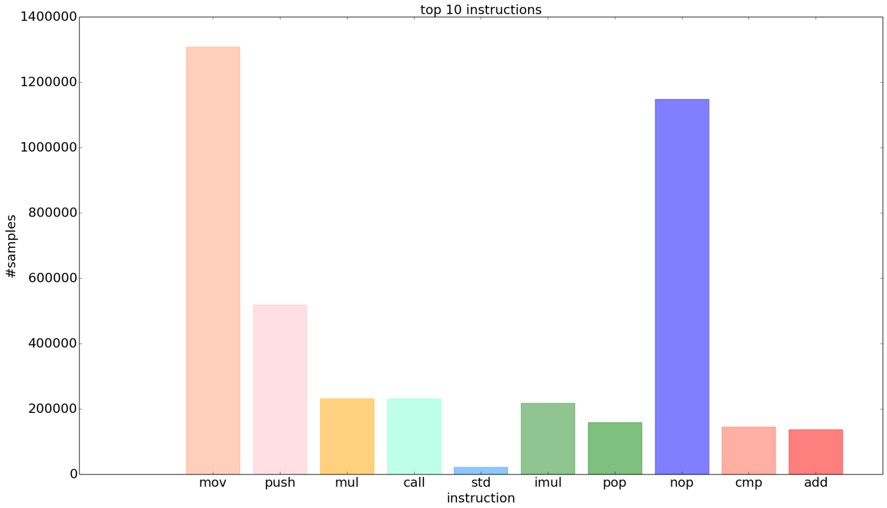


Figure 4.4: Top 10 opcodes in the training dataset

In addition, the average of instructions per each malware family is the following.

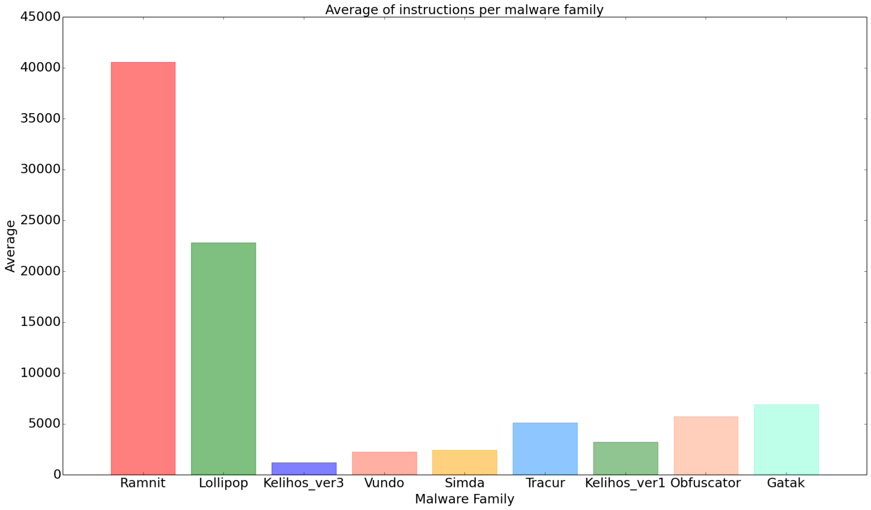


Figure 4.5: Average of opcodes per malware family

One particularity of the training dataset is that there are some malware samples that due to code obfuscation techniques do not have any instruction.

	#samples
Ramnit	0
Lollipop	2
Kelihos_ver3	4
Vundo	22
Simda	0
Tracur	0
Kelihos_ver1	6
Obfuscator.ACY	9
Gatak	0

Table 4.1: Number of samples per class with 0 instructions

2. Assembler directives or pseudo-ops. Are commands part of the assembler syntax but are not related to the x86 processor instruction set. All assembly directives start with a period (.).

   ```
   .data
   .bss
   ```

   The .data and .bss directives change the current section to .data or .bss, respectively.

3. Macros. Are basically a text substitution mechanism. A macro is a sequence of instructions, assigned by a name and could be used anywhere in the program. The syntax for a macro definition is:

   ```
   %macro macro_name  number_of_params
   <macro body>
   %endmacro
   ```

4.3 Winner's solution

The competition was won by a team of three people, Jiwei Liu and Xueer Chen from the University of Pittsburgh and Xiaozhou Wang from Redman Technologies Inc. Their solution relied mainly in the extraction of the three following features from malware.

1. Opcode 2,3 and 4-grams.
   - They counted the frequent 1-gram opcodes by selecting only the opcodes that appear more than 200 times in at least one asm file and they ended up with 165 features.
   - All possible 2-gram counts were included (27225 features)
   - The 3-gram and 4-gram counts which were greater than 100 in at least one asm file were also included (21285 and 22384 features, respectively)
2. Segment line count. They counted the number of lines per section in the asm file and they also counted the number of different sections in all malware samples which curiously was 448 a number much greater than 9, the number of sections in which an asm is usually divided. That's because of the application of metamorphic and polymorphic techniques.
3. Asm file pixel intensity features. Instead of representing the bytes file as pixels they read the asm file as a binary file. They found that the first 800 pixel intensities were very useful features.

Then, they used XGBoost(a machine learning library focused on gradient boosted trees) and cross-validation to find the subset of features that best classified the malware in classes and they finally obtained the lowest public and private logloss of 0.003082695 and 0.002833228, respectively. In the next figure you can find the confusion matrix of the training data.

	Ramnit	Lollipop	Kelihos_ver3	Vundo	Simda	Tracur	Kelihos_ver1	Obfuscator.ACY	Gatak
Ramnit	1541	0	0	0	0	0	0	0	0
Lollipop	1	2476	0	0	0	1	0	0	0
Kelihos_ver3	0	0	2942	0	0	0	0	0	0
Vundo	0	0	0	475	0	0	0	0	0
Simda	2	0	0	0	39	1	0	0	0
Tracur	1	0	0	0	0	750	0	0	0
Kelihos_ver1	0	0	0	0	0	0	398	0	0
Obfuscator.ACY	0	0	1	0	0	0	0	1225	2
Gatak	0	1	0	0	0	0	0	5	1007

Table 4.2: Winner's solution: confusion matrix

where they classified correctly 10854 of 10868 samples $(0,9987\%)$.

4.4 Novel Feature Extraction, Selection and Fusion for Effective Malware Family Classification

In [1] they presented an approach that extracts and combines different characteristics from malware and their fusion according to a perclass weighting paradigm. Their method achieved an accuracy of 0.998% on the Microsoft Malware Challenge dataset but what is more interestingly about their approach is that they also extracted feature patterns from images of malware.

Next you will find the different extracted set of features.

1. N-Gram (1G and 2G):
   1-Gram and 2-Gram features from the hexadecimal representation of binary files described with a 256-dimensional vector and 256^2-dimensional vector for the 1-Gram and 2-Gram models, respectively.

2. Metadata (MD1 and MD2):
   They extracted the size of the file and the address of the first byte sequence.

3. Entropy (ENT):
   Entropy can be defined as a measure of the amount of the disorder and it is used to detect the presence of obfuscation in malware files and for this reason they computed the entropy of all the bytes in a malware file.

4. Image patterns (IMG1 and IMG2):
   They extracted the Haralick features and the Local Binary Patterns features from each malware sample represented as a gray-scale image.

5. String length (STR):
   They extracted possible ASCII strings and its length from each PE using its hex dump.

6. Symbol frequencies (SYM):
   The frequencies of the symbols -, +, *,], [, ?, @ are extracted from the disassembled files because are typical of code designed to evade detection by resorting to indirect calls or dynamic library loading.

7. Operation code (OPC):
   They counted the number of times a subset of 98 operation codes appeared in each disassembled file. These subset was selected based either on their commonness or on their frequent use in malicious applications.

8. Register (REG):
   They computed the frequency of use of the registers in x86 architecture.

9. Application Programming Interface (API):
   They measured the frequency of use of the top 794 frequent APIs used in malware binaries based on the analysis performed in `https://www.bnxnet.com/top-maliciously-used-apis/`.

10. Section (SEC):
   They extracted different characteristics from sections in the disassem-

bly files such as the total number of lines in .bss, .txt, .data, etc sections or the proportion of lines in each section compared to the whole file.

11. Data Define (DP):
    They computed the frequency of db, dw and dd instruction because there are malware samples that do not contain any API call and only contain few operation codes, because of packing.

12. Miscellaneous (MISC):
    This features are composed by the frequency of 95 keywords manually chosen from the disassembled code.

Following you will find a table containing the list of feature categories and their evaluation with XGBoost.

Feature Category	#Features	Accuracy	Logloss
ENT	203	0.9987	0.0155
1G	256	0.9948	0.0307
STR	116	0.9877	0.0589
IMG1	52	0.9718	0.1098
IMG2	108	0.9736	0.1230
MD1	2	0.8547	0.4043
MISC	95	0.9984	0.0095
OPC	93	0.9973	0.0146
SEC	25	0.9948	0.0217
REG	26	0.9932	0.0352
DP	24	0.9905	0.0391
API	796	0.9905	0.0400
SYM	8	0.9815	0.0947
MD2	2	0.7655	0.6290

Table 4.3: List of feature categories and their evaluation in XGBoost

After the feature extraction process, they combined the features using a version of the forward stepwise selection algorithm. The original version of this algorithm starts with a model containing no features and then gradually increases the feature set by adding one feature at each iteration. Instead of considering one feature at a time, they added all the subset of features belonging to a feature category at a time, until when adding more features didn't increase the value of logloss. By combining the feature categories as described earlier, they achieved a test logloss of 0.0063 positioning its solution among the top 10 in the competition.

4.5 Deep Learning Frameworks

Deep Learning is a hot field in Artificial Intelligence and Machine Learning, and thus, there are various deep learning libraries available open-source. The most popular are:

1. Caffe.
   It is a python deep learning framework developed by the Berkeley Vision and Learning Center. It allows you to define if train using the CPU or the GPU easily. Caffe benefits from having a huge repository with pre-trained neural network models suited for many problems. It has a great implementation for convolutional networks but it has no implementation for recurrent networks.

2. Theano.
   It is a python deep learning library which make use of symbolic graph for programming the networks. It also allows you to visualize the computation graphs with d3viz.

3. TensorFlow.
   It is written with a Python API over a C/C++ engine that makes it run fast. It is more than a deep learning framework, and it has tools to support reinforcement learning and other algorithms. In addition, TensorFlow can also be deployed in phones thanks that it can be compiled in ARM architectures.

4. Deeplearning4j.
   It is a deep learning framework developed in Java. It aims to be the scikit-learn library in the deep learning space.

5. Torch.
   It is a computational framework written in Lua that supports machine learning algorithms. It has been used by large scale companies such as

Google and Facebook. However, it is not as well-documented as other deep learning frameworks.

TensorFlow has been chosen mainly because it has a Python API, there's a lot of documentation available and it has a large community that it continuously develops the library. In addition, it is very easy to setup and to learn and recently, they released TensorBoard, a tool to visualize TensorFlow graphs and to plot some metrics such as the accuracy or the loss at each training iteration. Moreover, it provides support for distributed computing since version 0.8 (currently 0.11).

Chapter 5

Learning Feature Extractors from Malware Images

The problem of malware detection and classification is a very complex task and there's no perfect approach to tackle it. For this reason AV vendors rely in hybrid approaches that make use of traditional signature-based, heuristic-based and machine learning methods as well as human analysis.

This chapter presents a novel approach for malware classification based on the work performed by Nataraj et al. [21] which introduced the idea of representing malicious software as gray-scale images. Then, they extracted different features using GIST and they used the k-nearest neighbor algorithm for classification. Our approach differ in the point that we use Convolutional Neural Networks for learning discriminative patterns from the malware images.

The next sections explain how malware can be visualized as images followed by the architectures of the different CNNs tested and its specifications as well as the results obtained in the Kaggle's competition.

5.1 Visualizing malware as gray-scale images

This thesis is highly motivated by the work in [21] which is based on the observation that images of different malware samples from the same family appear to be similar while images of malware samples belonging to a different family are distinct. Moreover, if old malware is re-used to create new malware binaries the resulting ones would be very similar visually.

In their work, they computed image based features to characterize malware. For that purpose, to compute texture features they used GIST [24]. The resulting feature vectors were used to train a K-nearest neighbor classifier with Euclidean distance. As introduced in [21], a given malware binary file can be read as a vector of 8 bit unsigned integers and organized into a 2D array. Then, this array can be visualized as a gray scale image in the range [0,255].

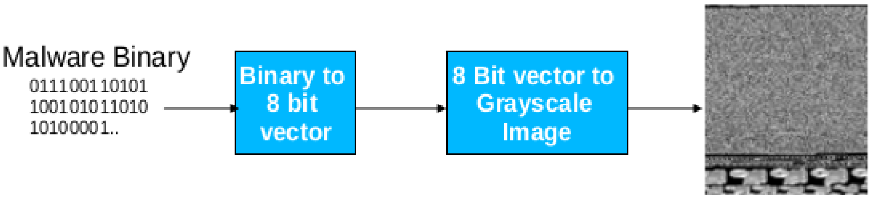


Figure 5.1: Visualizing Malware as a Gray-Scale Image

The main benefit of visualizing malware as an image is that the different sections of a binary can be easily differentiated. In addition, as malware authors only change a small part of the code to produce new variants, images are useful to detect small changes while retain the global structure. In consequence, malware variants belonging to the same family appear to be very similar as images while also being distinct from images of other families.

5.1.1 Malware families

Following are presented some malware files of each malware variant in the dataset. One particularity of the dataset is that the samples do not contain the PE header because it was removed to ensure sterility.

1. Ramnit. This type of malware is known to steal your sensitive information such as user names and passwords and it also can give access to an illegitimate user to your computer.

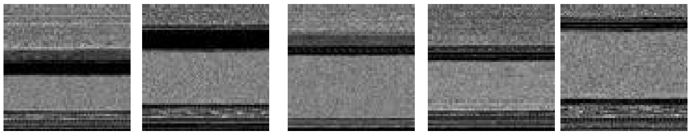


Figure 5.2: Rammit samples

2. Lollipop. This malware shows ads in your browser and redirects your search engine results. In addition, it tracks what you are doing on your computer. This type of malware usually is downloaded from the program's website or by some third-party software installation programs.

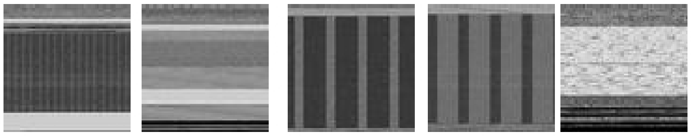


Figure 5.3: Lollipop samples

3. Kelihos_ver3. Third version of the Kelihos botnet. Kelihos is mainly involved in spamming and theft of bitcoins. This trojan can give access and control of your computer to an illegitimate user and can also communicate with other computers about sending spam emails, run malicious programs and steal sensitive information.

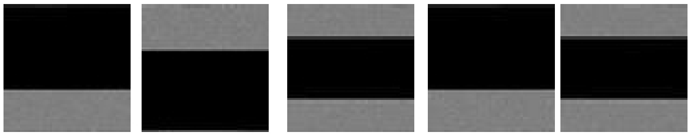


Figure 5.4: Kelihos_ver3 samples

4. Vundo. This trojan is known to cause popups and advertising for rogue antispyware programs. In addition, sometimes is used to perform denial of service attacks and also to deliver malware to other computers.

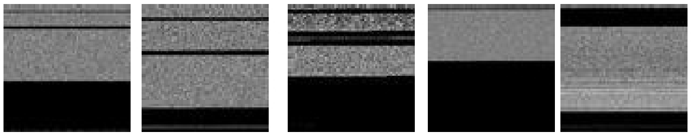


Figure 5.5: Vundo samples

5. Simda. It is a family of backdoors that try to steal sensitive information such as usernames, passwords and certificates via its keylogging and HTML injection routines. It also can give a hacker access to your computer.

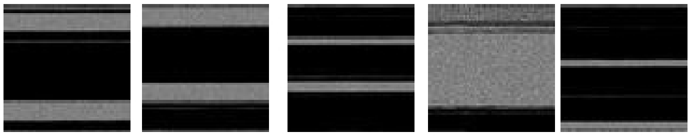


Figure 5.6: Simda samples

6. Tracur. This trojan hijacks results from different search engines such as google, youtube, yahoo, etc, and redirects to a different web page. It also give a hacker access to your computer and can be used to download other types of malware.

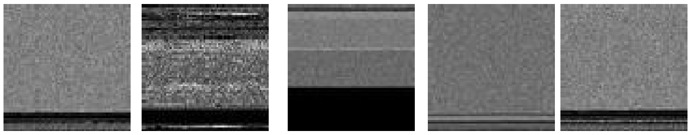


Figure 5.7: Tracur samples

7. Kelihos_ver1. First version of the Kelihos botnet. It was first discovered at the end of 2010 having infected 45.000 machines and sending about 4 billions spam messages per day.

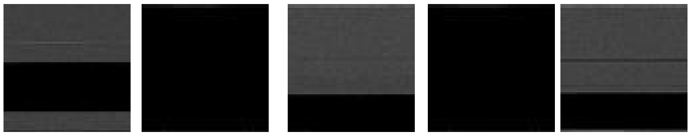


Figure 5.8: Kelihos_ver1 samples

8. Obfuscator.ACY. This class comprises all malware that has been obfuscated to hide their purposes and to not be detected. The malware that lies underneath this obfuscation can have almost any purpose.

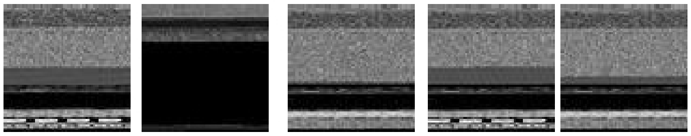


Figure 5.9: Obfuscator.DCY samples

9. Gatak. It is a trojan that gathers information about your pc and sends it to a hacker. It also downloads other malware files in your computer. This trojan is usually downloaded when downloading a key generator or a software crack.

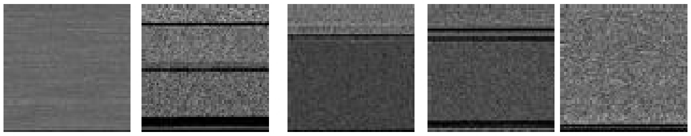


Figure 5.10: Gatak samples

5.2 CNN Architectures

In this thesis, we have proposed a novel approach to classify samples of malicious software from their representation as gray-scale images. In the work of [21] they used a traditional recognition approach to classify gray-scale images of malware. First of all they extracted texture features from the malware gray-scale images and then, they trained a K-NN classifier.

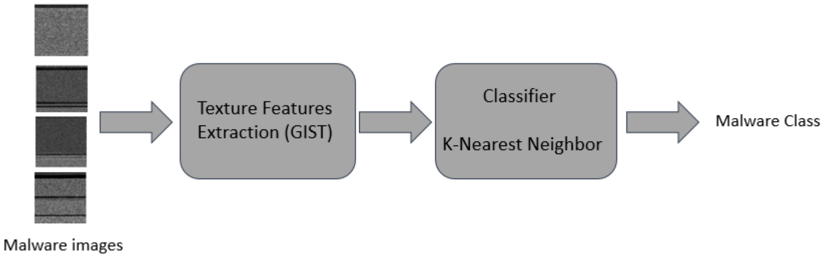


Figure 5.11: Shallow Approach

The main problem of their approach is that it doesn't scales well with lots of data. Accordingly, two ways of improvement are (1) keep building more features like SIFT, HoG, etc and (2) using another classifier like Random Forests or SVM. Instead, our approach makes use of Convolutional Neural Networks to learn a feature hierarchy all the way from pixels to the layers of the classifier.

This section presents the different architectures of the network and its specifications. The details of the architectures are defined in figures 5.12, 5.13 and 5.14.

All architectures have in common the input and the output layers. On one hand, the input layer consists of N neurons, being N the size of the training images. The image and the height of the images varies depending on the file size and thus, before feeding the images as input all images had been downsampled to 32 by 32 pixels. In consequence, N is equals to $32*32 = 1024$. On the other hand, all architectures have an output layer of 9 neurons because the architectures are designed to handle a 9-class classification problem. In addition, after each densely-connected layer it was applied dropout to reduce overffiting.

To determine the parameters for each architecture it was performed a grid search. Specifically, the grid search was used to determine the optimum learning rate, the size of the kernels of each convolutional layer and the number of filters applied and also the number of neurons in each densely-connected layer. Finally, to reduce the search space some parameters were fixed such as the mini-batch size to 256, the region of the max-pooling layer to 2x2 with stride equals to 2 and the learning rate to 0.001

5.2.1 CNN A: 1C 1D

The architecture consists of:

1. Input layer of NxN pixels (N=32).

2. Convolutional layer (64 filter maps of size 11x11).

3. Max-pooling layer.

4. Densely-connected layer (4096 neurons)

5. Output layer. 9 neurons.

The input layer consists of 32x32 neurons and is followed by a convolutional layer composed by 64 filters of size 11x11. The output of the convolutional layer is $(32-11+1)*(32-11+1) = 22*22 = 484$ for each feature map. As a result, the total output of the convolutional layer is $22*22*64 = 30976$. After that, the pooling layer takes the output of each feature map from the convolutional layer and outputs the maximum activation of all 2x2 regions. In consequence, the output of the pooling layer is reduced to $11*11*64 = 7744$. The pooling layer is then followed by a fully-connected layer with 4096 neurons and every neuron of this layer is also connected to each one of the neurons in the output layer.

The number of learnable parameters P of this network is:

$$P = 1024*(11*11*64)+64+(11*11*64)*4096+4096+4096*9+9 = 39690313$$

where $(11*11*64)+64$ are the shared weights for every feature map and 64 is the total number of shared bias.

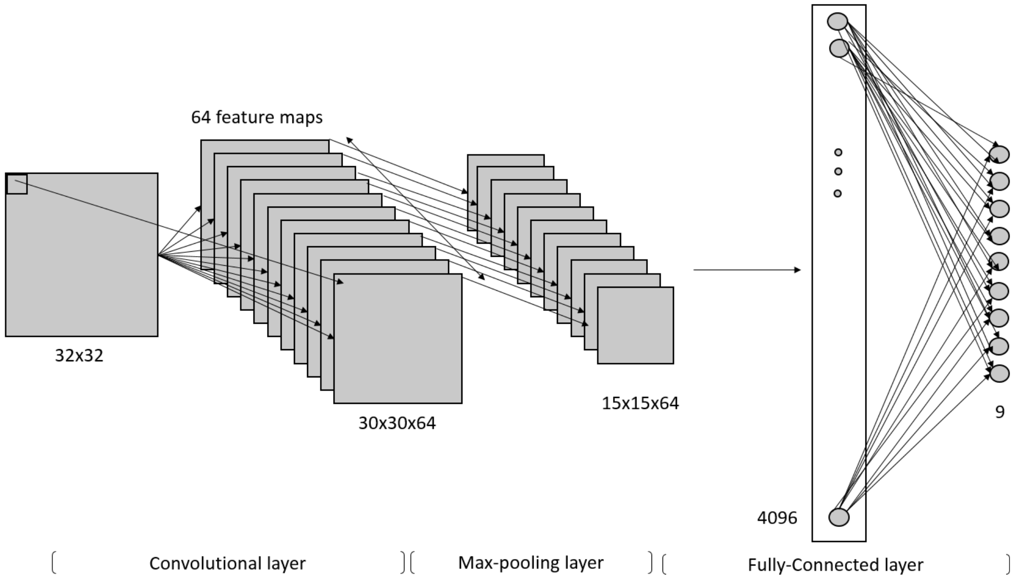


Figure 5.12: Overview architecture A: 1C 1D

5.2.2 CNN B: 2C 1D

The architecture consists of:

1. Input layer of NxN pixels (N=32).
2. Convolutional layer (64 filter maps of size 3x3).
3. Max-pooling layer.
4. Convolutional layer (128 filter maps of size 3x3).
5. Max-pooling layer.
6. Densely-connected layer (512 neurons).
7. Output layer. 9 neurons.

As in the previous architecture, the input layer consists of 32x32 neurons and is followed by a convolutional layer composed by 64 filters of size 3x3. The output of the convolutional layer is $(32-3+1)*(32-3+1) = 30*30 = 900$ for each feature map and a total of $30*30*64 = 57600$. Next it is applied a max-pooling layer which takes as input the output of the convolutional layer and outputs the maximum activation of all 2x2 regions reducing the output to $15*15*64$. Then, the pooling layer is followed by another convolutional layer of 128 filters with 3x3 receptive fields. After the convolutional layer follows another pooling layer that takes as input the output of the previous convolutional layer that is 13*13*128 and reduces the output to 7*7*128. Finally, the polling layer is followed by a densely-connected layer with 512 neurons.

The number of learnable parameters P of this network is:

$$P = 1024*(3*3*64)+64+(15*15*64)*(3*3*128)+128+$$

$$+(7*7*128)*512+512+512*9+9 = 20395209$$

where $(3*3*64)+64$ and $(3*3*128)+128$ are the shared weights for every feature map and 64 and 128 are the number of shared bias in the first and second convolutional layers,respectively.

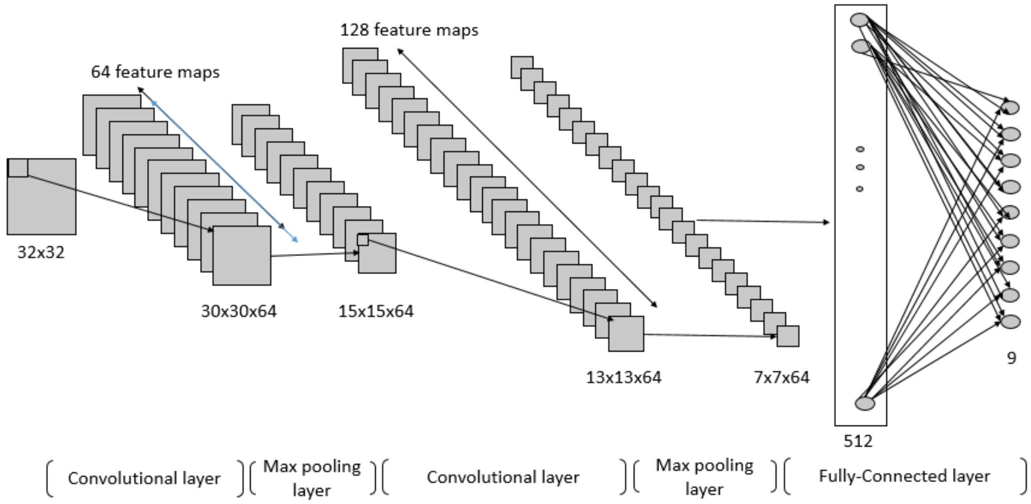


Figure 5.13: Overview architecture B: 2C 1D

5.2.3 CNN C: 3C 2D

The architecture consists of:

1. Input layer of NxN pixels (N=32).

2. Convolutional layer (64 filter maps of size 3x3).

3. Max-pooling layer.

4. Convolutional layer (128 filter maps of size 3x3).

5. Max-pooling layer.

6. Convolutional layer (256 filter maps of size 3x3).

7. Max-pooling layer.

8. Densely-connected layer (1024 neurons).

9. Densely-connected layer (512 neurons).

10. Output layer. 9 neurons.

It starts with an input layer with 32x32 neurons which is then followed by a convolutional layer with 64 filters of size 3x3. The output of the convolutional layer is 30x30x64 and is used to feed the following max-pooling layer that reduces its input to 15x15x64. Next follows the second convolutional layer with 128 filters of size 3x3. After the convolutional layer it follows the second pooling layer that takes as input the output of the second convolutional layer ($13 * 13 * 128$) and outputs 128 feature maps of size 7x7. Moreover, a third convolutional layer with 256 filters of size 3x3 follows the second pooling layer which outputs 256 feature maps of size 5x5. Additionally, a third pooling layer follows the convolutional layer reducing the input to 256 feature maps of size 3x3. Lastly, follows two densely-connected layers of 1024 and 512 neurons, respectively.

The number of learnable parameters P of this network is:

$$P = 1024{*}(3{*}3{*}64){+}64{+}(15{*}15{*}64){*}(3{*}3{*}128){+}128{+}(7{*}7{*}128){*}(3{*}3{*}256){+}$$

$$+256 + (3 * 3 * 256) * 1024 + 1024 + 1024 * 512 + 512 + 512 * 9 + 9 = 34519497$$

where $(3*3*64)+64$, $(3*3*128)+128$ and $(3*3*256)+256$ are the shared weights for every feature map and 64 and 128 are the number of shared bias of the first, second and third convolutional layers, respectively.

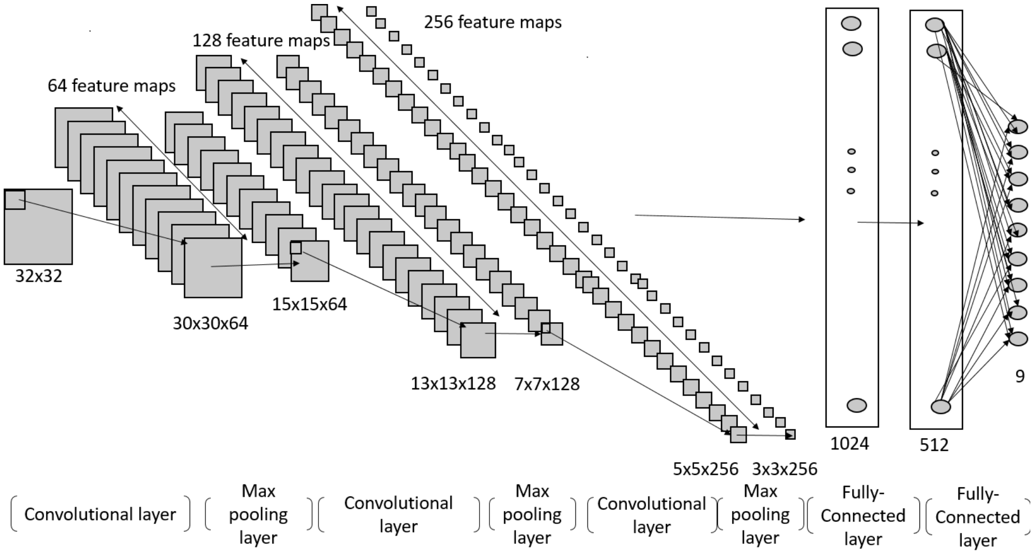


Figure 5.14: Overview architecture C: 3C 2D

5.3 Results

The content of this section is structured as follows. First are presented the results of the CNNs obtained during training and validation and then, are presented the scores achieved in the competition.

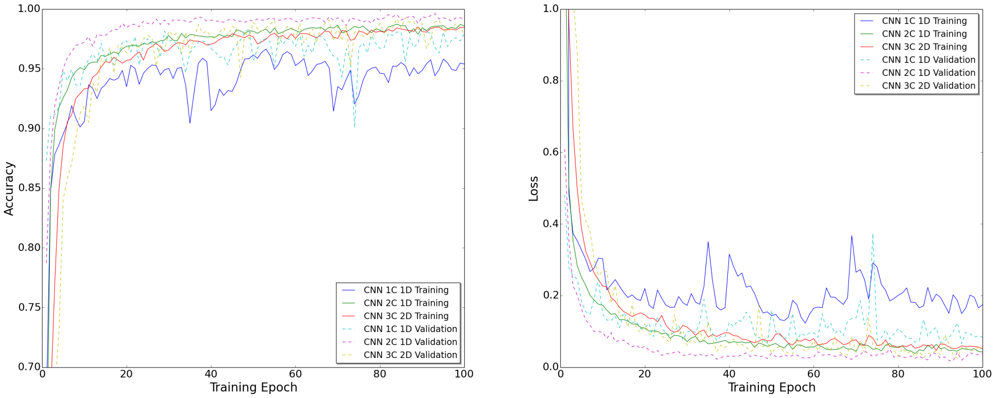
5.3.1 Evaluation

The dataset provided by Kaggle for training was divided into two:

1. The training set of size $(N - N/10) = 9781$
2. The validation set of size $M = N/10 = 1086$

where N is the total size of the dataset, $N = 10868$ and $M = 1086$. The validation set was used to search the parameters of the networks and to know when to stop training. In particular, we stopped training the network if the validation loss increased in 10 iterations.

The next figure shows the accuracy and the cross-entropy achieved by the models presented in 5.2 until they reached the 100th training iteration.



(a) Training & Validation accuracy (b) Training & Validation Cross-Entropy

Figure 5.15: Approach A: CNNs training results

It can be observed that the performance of the CNN with only one convolutional layer performs poorly than the other nets. Next you will find the performance of the networks on the training set at the 500th iteration.

- CNN 1C 1D. Accuracy: 0.9857 Cross-entropy: 0.0968

	Rammit	Lollipop	Kelihos_ver3	Vundo	Simda	Tracur	Kelihos_ver1	Obfuscator.ACY	Gatak
Rammit	1534	0	0	0	1	0	0	1	5
Lollipop	0	2375	0	0	0	4	0	0	98
Kelihos_ver3	0	1	2937	0	0	0	0	0	4
Vundo	0	0	0	472	0	0	2	0	1
Simda	0	0	0	1	41	0	0	0	0
Tracur	2	0	0	2	0	737	0	4	10
Kelihos_ver1	0	0	0	0	0	0	387	0	11
Obfuscator.ACY	0	0	0	0	0	1	0	1219	8
Gatak	0	0	0	0	0	2	0	0	1011

Table 5.1: CNN 1C 1D: confusion matrix

- CNN 2C 1D. Accuracy: 0.9976 Cross-entropy: 0.0231

	Rammit	Lollipop	Kelihos_ver3	Vundo	Simda	Tracur	Kelihos_ver1	Obfuscator.ACY	Gatak
Rammit	1539	0	0	0	0	0	0	1	1
Lollipop	0	2471	0	0	0	1	0	0	5
Kelihos_ver3	0	0	2938	0	0	0	4	0	0
Vundo	0	0	0	474	0	0	0	0	1
Simda	0	0	0	1	41	0	0	0	0
Tracur	0	0	0	0	0	750	0	0	1
Kelihos_ver1	0	0	0	0	0	0	394	0	4
Obfuscator.ACY	0	1	0	0	0	2	2	1223	0
Gatak	0	0	0	0	0	0	0	0	1013

Table 5.2: CNN 2C 1D: confusion matrix

- CNN 3C 2D. Accuracy: 0.9938 Cross-entropy: 0.0257

	Rammit	Lollipop	Kelihos_ver3	Vundo	Simda	Tracur	Kelihos_ver1	Obfuscator.ACY	Gatak
Rammit	1533	0	0	0	0	1	0	5	2
Lollipop	0	2443	0	0	0	1	0	0	33
Kelihos_ver3	0	0	2938	0	0	0	4	0	0
Vundo	0	0	0	474	0	0	0	0	1
Simda	0	2	0	0	39	0	0	1	0
Tracur	0	0	0	0	0	747	1	0	3
Kelihos_ver1	0	0	0	0	0	0	393	0	4
Obfuscator.ACY	0	3	0	0	0	1	3	1211	3
Gatak	0	0	0	0	0	0	0	0	1013

Table 5.3: CNN 3C 2D: confusion matrix

It can be observed that the convolutional neural networks with one and three convolutional layers had problems mainly while labeling samples from Rammit, Lollipop, Tracur, Kelihos_ver1 and Obfuscator.ACY and they ended up misclassifying some samples as belonging to the Gatak malware's family. In particular, the major number of misclassifications had been produced from samples of the Lollipop family, with 98 and 33 incorrect classifications from the convolutional net with one and three convolutional layers, respectively. Moreover, it can be seen that the training error of the convolutional network with two layers is lower than the other two because it greatly reduced the number of samples misclassified as Gatak and it achieved a training accuracy of 0.9978 very near to the one obtained by the winner's solution (0.9987) and a loss of 0.0231 which is also lower the obtained in [1] using only the subset of features named IMG1 (Haralick features) and IMG2 (Local Binary Pattern features) as represented in 4.3 which is 0.9718 & 0.1098 and 0.9736 & 0.1230, respectively.

5.3.2 Testing

Usually, Kaggle provides a test set without label in their competitions and the Microsoft Malware Classification Challenge is not different. Therefore, to evaluate our models using the test set we have to submit a file with the predicted probabilities for each class to Kaggle. These submissions are evaluated using the multi-class logarithmic loss. The logarithmic loss metric is defined as:

$$logloss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{i,j} log(p_{i,j})$$

where N is the number of observations, M is the number of class labels, log is the natural logarithm, $y_{i,j}$ is 1 if the observation i is in class j and 0 otherwise, and $p_{i,j}$ is the predicted probability that observation i is in class j.

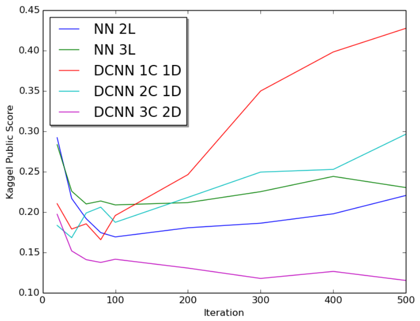
This type of evaluation metric provides extreme punishment for being confident and wrong. That is, if the algorithm makes a single prediction that an observation is definitely true (1) when it is actually false, it adds infinity to the error score making every other observation pointless. Hence, in their competitions, Kaggle bound the predictions away from extremes by using the following formula:

$$logloss = -\frac{1}{N}\sum_{i=1}^{N}(y_i log(p_i) + (1 - y_i) log(1 - p_i))$$

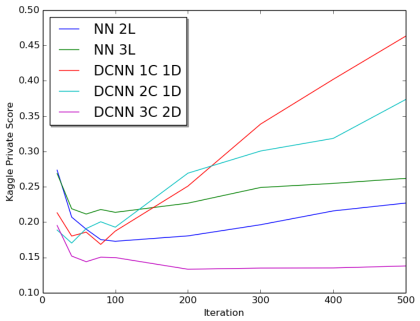
Moreover, the submitted probabilities are not required to sum to one because they are rescaled prior to being scored.

Additionally, submissions in Kaggle are evaluated with two scores, the public score and the private score where the first one is calculated on approximately

30% of the test data and the second one is calculated on the other 70%.



(a) Public score



(b) Private score

The best results where obtained by the CNN with 3 convolutional layers and 2 densely-connected layers which obtained a public and a private score at iteration 500 of 0.117629734 and 0.134821767, respectively. That is an improvement of 94.64% and 93.86% with respect to the equal probability benchmark (logloss=2.197224577) which is obtained by submitting 1/9 for every prediction. In contrast, other models achieved their respective lowest score between iteration 50 and 100 which coincide with the point where the algorithm converges into a local minima but unfortunately they were not able to learn a better underlying relationship on the training data and ended up performing much worse than the convolutional network with 3 convolutional layers.

Chapter 6

Convolutional Neural Networks for Classification of Malware Disassembly Files

As described in sections 4.3 and 4.4, the approaches that performed better in the competition where those that extracted features from the disassembled files such as n-grams counts. A n-gram is a contiguous sequence of n items from a sentence. In our case, those items are opcodes extracted from the disassembled files. However, the main problem that has extracting those n-grams is that the number of features extracted increases exponentially as N increases. In particular, a 2-gram model will result in a two-dimensional matrix of size $256^2 = 65536$, a 3-gram model will result in a three-dimensional matrix of $256^3 = 16777216$ features, a 4-gram model in a four-dimensional matrix of $256^4 = 4294967296$ and so on which turns out to be very computationally expensive.

In this thesis, it is used the CNN architecture introduced by Yoon Kim in [12] as an alternative to n-grams because is much more computationally efficient.

The network trained by Yoon Kim was a simple CNN with one layer on top of word vectors obtained using Word2Vec, an unsupervised neural language model. These word vectors were trained using the model proposed by Mikolov in [20] on 100 billion words of Google News avaliable in `https://code.google.com/p/word2vec/`.

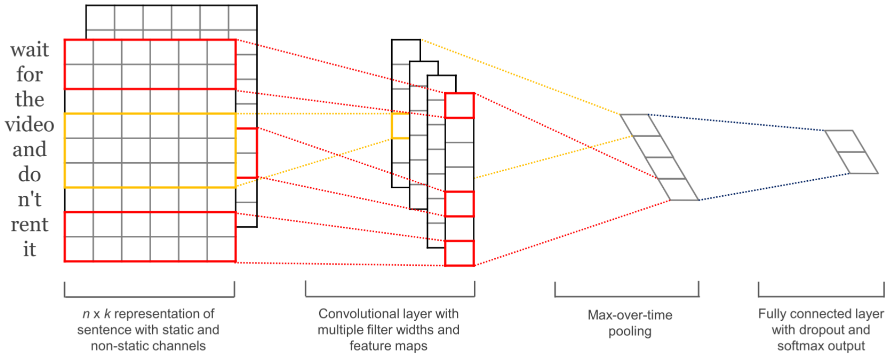

wait
for
the
video
and
do
n't
rent
it
n x k representation of sentence with static and non-static channels
Convolutional layer with multiple filter widths and feature maps
Max-over-time pooling
Fully connected layer with dropout and softmax output

Figure 6.1: Yoon Kim model architecture

The structure of this chapter is organized as follows. First, it is introduced what are word embeddings and in particular, the Skip-Gram model. Then, it is described the architecture of the CNN and lastly, the results obtained are compared against the best solutions from Kaggle's competition.

6.1 Representing Opcodes as Word Embeddings

Traditionally, Natural Language Systems treated words as symbols, creating encodings that are randomly generated and do not provide useful information for the system regarding the relationship among symbols. For example, "The" may be represented "Id156". By using vector representations of words these problems are solved. There are various techniques of language modeling that allow to map words or phrases from vocabulary, let's say the English language, to vectors of real numbers in a low-dimensional space. These set of language models differ in the way they learn vector representation of words but are all based on the distributional hypothesis which considers that words that are used and occur in the same contexts tend to share semantic meaning.

The approaches that are based on this hypothesis can be divided broadly into two categories:

- Count-based methods. They compute how often some word co-occurs with its neighbors in a corpus and then they map these count-based statistics to a vector for each word.
- Predictive methods. They try to predict a word given its neighbors in terms of learned small, dense embedding vectors.

Word2Vec [20] is a very efficient predictive model for learning word embeddings. Word2Vec comes with two different approaches to learn the vector representations of words: (1) the Continuous Bag of Words (CBOW) and (2) the Skip-Gram model. The main difference between both models is that CBOW predicts target words from source context words while the skip-gram model predicts source context words from target words (the context of a word are the words to the left of the target and the words to the right of the target).

To learn the word embeddings we used the Skip-Gram approach and thus, an explanation of the CBOW model is not provided because it is out of scope.

6.1.1 Skip-Gram model

The Skip-Gram model tries to predict each context word from its target word. Thus, the input to the model is w_i and the output is w_{i-2},w_{i-1},w_{i+1},w_{i+2} if a window size of 2 is used.

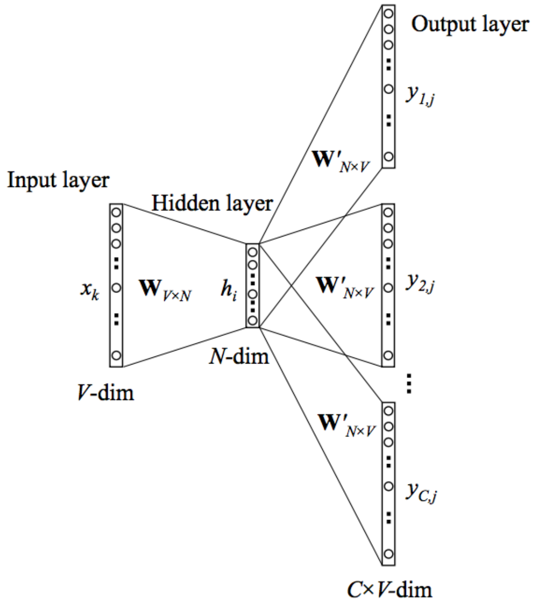


Figure 6.2: Skip-gram model architecture

As the network can't be feed with words just as text strings is needed a way to represent words. For that purpose, first it is build a vocabulary of words from the malware training samples. In the case all operation codes appear in the samples the vocabulary will consist of 665 words. Accordingly, a word like "push" is going to be represented as a one-hot vector. This vector will have 665 components (one for every word in the vocabulary) and in the position corresponding to the word "push" it will place a 1 and 0s in all of the other positions.

The output layer depends on the window size. Thus, for a window size one (just predicting one word to the left and to the right of the targets word) the network will output a two-dimensional vector, with one dimension of the vector containing the probabilities of the words in the vocabulary to appear at the left of the target word and the other dimension containing the probabilities of the words in the vocabulary to appear at the right of the target word. The dimension of the hidden layer or embedding layer corresponds to $V * E$, where V is the size of the vocabulary and E is the embedding size.

The training objective of the Skip-gram model is to find word representations that are useful for predicting surrounding words in a corpora. Formally, given a sentence of words $w_1, w_2, ..., w_T$ the objective is to maximize the average log probability defined as:

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c, j\neq 0} log\, p(w_{t+j}|w_t)$$

where c is the size of the training context (Larger c results in more training examples and can lead to a higher accuracy at the expense of the training time) and $p(w_o|w_I)$ is formulated as:

$$p(w_o|w_I) = \frac{exp({v_{w_o}}^T v_{w_I})}{\sum_{w=1}^{W} exp({v_w}'^T v_{w_I})}$$

where v_w and $v_w{}'$ are the "input" and "output" vector representations of words in the vocabulary and W is the number of words in the vocabulary.

The main drawback of this formulation is that as W is often large (10^5) the cost of computing $\bigtriangledown log\, p(w_{t+j}|w_t)$ is impractical. An alternative of the full softmax is the Noise-Contrastive Estimation (NCE). NCE postulates that if a model is able to differentiate data from noise by means of logistic regression is a good model. NCE can be shown to approximately maximize the log probability of the softmax. The basic idea behind NCE is to train a logistic regression classifier to discriminate between samples from the data distribution and samples from the "noise" distribution, based on the ratio of probabilities under the model and the noise distribution. Under those circumstances, as the Skip-Gram model is only concerned with learning high-quality vector representations the NCE can be simplified as:

$$log\, \theta({v_{w_o}}^T v_{w_I}) + \sum_{i=1}^{K} \mathbb{E}w_i \sim P_n(w)[log\, \theta(-{v_{w_o}}^T v_{w_I})]$$

which is used to replace every $log(w_o|w_I)$ term in the Skip-Gram objective. In consequence, the task is to distinguish the target word w_o from draws of the noise distribution $P_n(w)$ using logistic regression, where there are K negative samples for each data sample ($K \simeq 5 - 20$). The noise distribution $P_n(w)$ is a design parameter. It was selected the unigram distribution $U(w)$ of the training data as the noise distribution because it is known to work well for training language models. This distribution assumes that each word in a sequence is independent and thus, each value would be independent of the other values. In consequence, we would need to estimate the probability of a sequence S in the malware's language model P(S|M). The probability generated for a specific sequence is calculated as follows: $P(S) = \prod_w^S P(w)$ To find the word embeddings it was used a window size equals to 5, meaning that for each target word, the skip-gram approach tried to predict the five words to the left and to the right. Following you will find the visualization of the learned embeddings, using the t-SNE algorithm. [35]

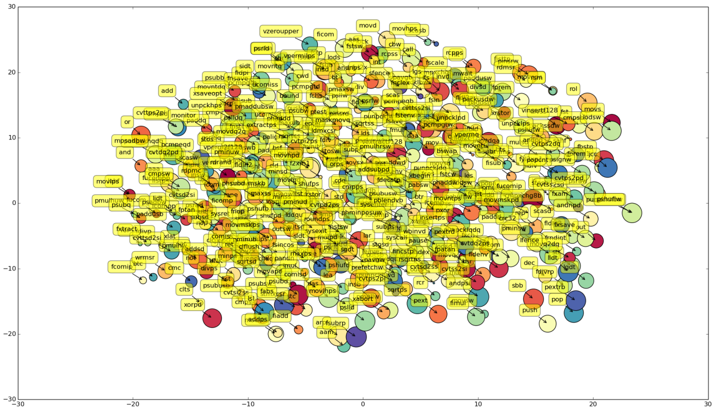


Figure 6.3: t-SNE representation of the word embeddings

For instance, the opcodes whose vector representations are most similar to the opcode "push" are:

1. pop

2. insertps

3. fucomp

which makes sense because tons of push instructions in malware files are followed by the pop instruction or viceversa and are the two opcodes most used. To compute the similarity between two vectors p and q it was used the Euclidean distance.

$$d(p,q) = d(q,p) = \sqrt{\sum_{i=1}^{n}(q_i - p_i)^2}$$

6.2 Convolutional Neural Network Architecture

The input of the network is a window of N (default=10000) opcodes from the disassembly file represented as one-hot vectors of size V. As the number of opcodes in every file is different we just extracted the first 10000 opcodes of each file. The criteria used to select N is the following:

- A lower N will result in not capturing enough information of malware samples belonging to the classes Ramnit and Lollipop which the average number of instructions per sample is greater than 20000.
- A higher N will result in an increase in the time needed for training the network.
- 10000 is greater than the average of opcodes of almost all malware samples except from those belonging to the Ramnit and Lollipop family (figure 4.5).

In addition, all malware files with less than N opcodes will be filled with UNKNOWN tokens ("UNK").

Regarding the embedding layer, is composed by a V x E matrix and it is the responsible of mapping the opcodes into low-dimensional vector representations. It's essentially a lookup table that can be learned during training or it can be initialized using the vector representations learned using any language model such as the Skip-Gram approach explained in section 6.1.1.

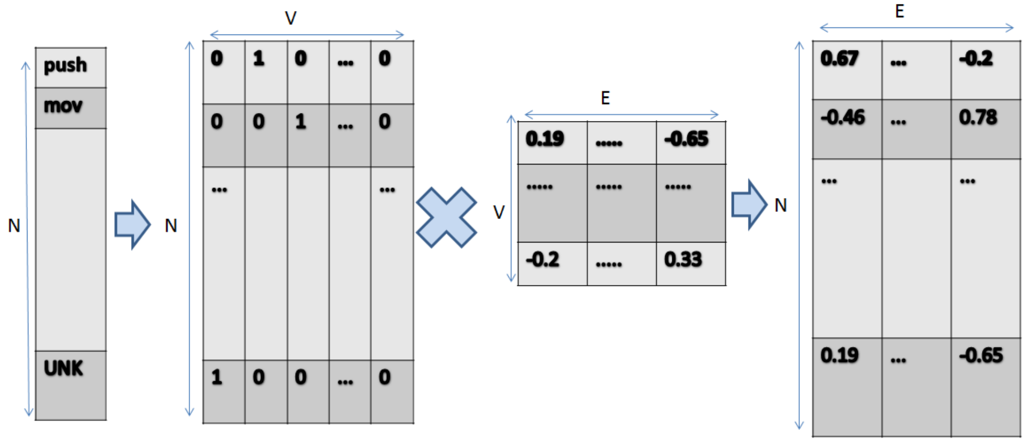


Figure 6.4: CNN Embedding layer output

The embedding layer is followed by a convolutional layer which have K (default 64) filters of different sizes (i x E, where *i* is an integer value representing the number of opcodes it covers. Each filter slides over the whole embedded input, but varies in how many opcodes it covers.

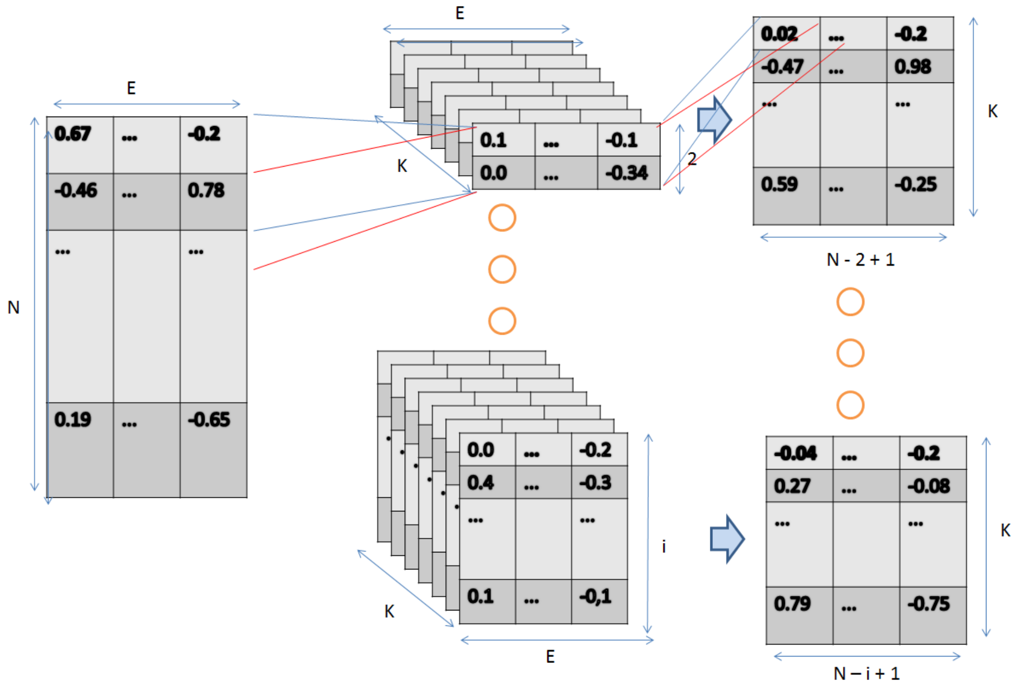


Figure 6.5: CNN Convolutional layer output

Following the convolutional layer it is performed max-pooling over the output of a specific filter size and leaves us with an output of size K, which is essentially a feature vector. Then, the output of the max-pooling layer is combined into one long feature vector. Lastly, the features obtained in the previous layer (with dropout applied) are used to generate the desired predictions by doing a matrix multiplication and selecting the class with the highest score.

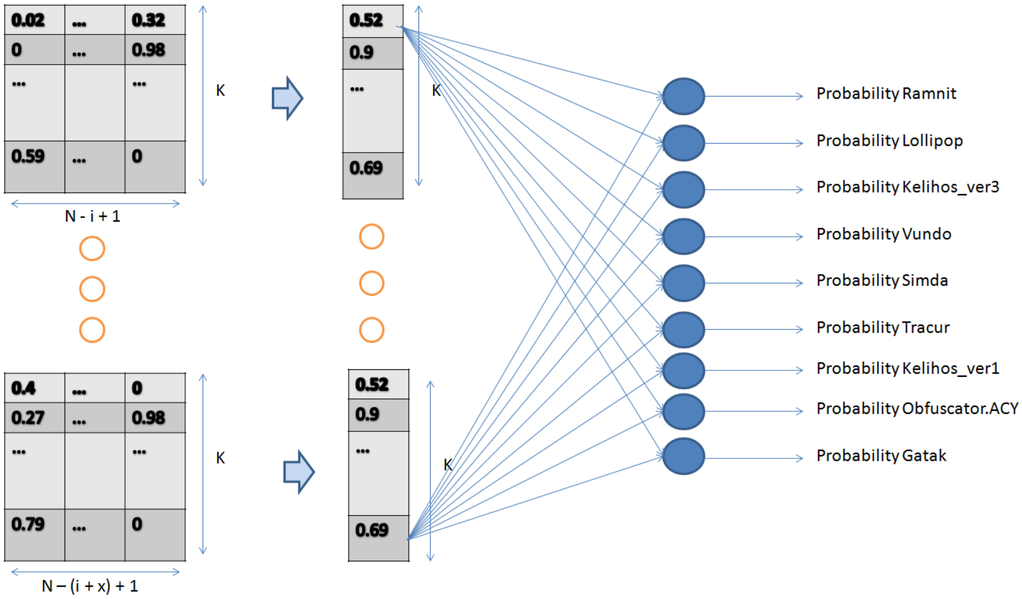


Figure 6.6: CNN Max-pooling & output layer

6.3 Results

Next, it is detailed how the parameters of the network were selected and the heuristic search that it was performed followed by the results obtained in both the training and the test set.

6.3.1 Evaluation

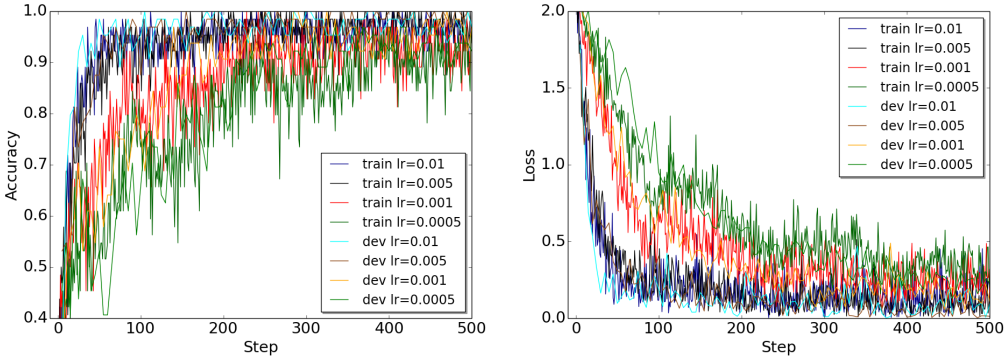
Heuristic Search

The parameters of the network were found by performing an heuristic search. First, it was selected the learning rate. After that, it was selected the embedding size E followed by the number of filters K and the different filter sizes. By default, the network was initialized with the following parameters:

1. Learning rate = 0.001
2. Embedding size = 32
3. #filters = 64
4. Filter sizes = [3,4,5]
5. Batch size = 64

Next you will find the values that were considered for each particular parameter of the network:

1. Learning rate: 0.01, 0.005, 0.001, 0.0005

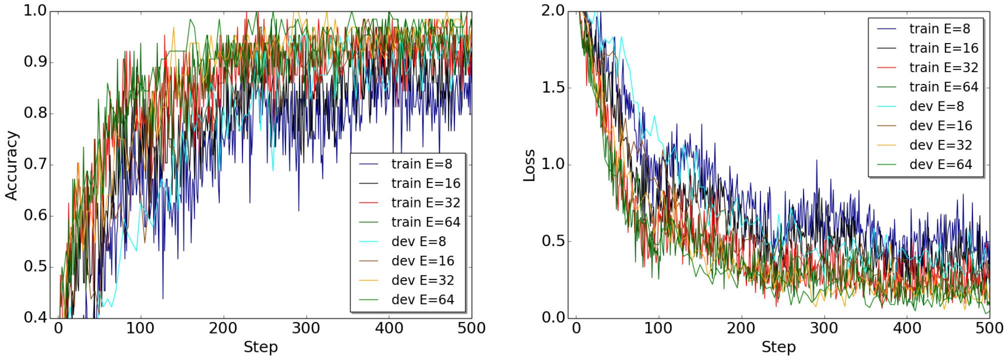


(a) Accuracy Comparison (b) Cross-entropy Comparison

Figure 6.7: Heuristic Search: Learning Rate

It was selected a $\lambda = 0.001$ because a large learning rate can make the gradient descent to overstep the minimum and also it has been tested in scientific publications that the value chosen works really well.

2. Embedding size: 8, 16, 32, 64

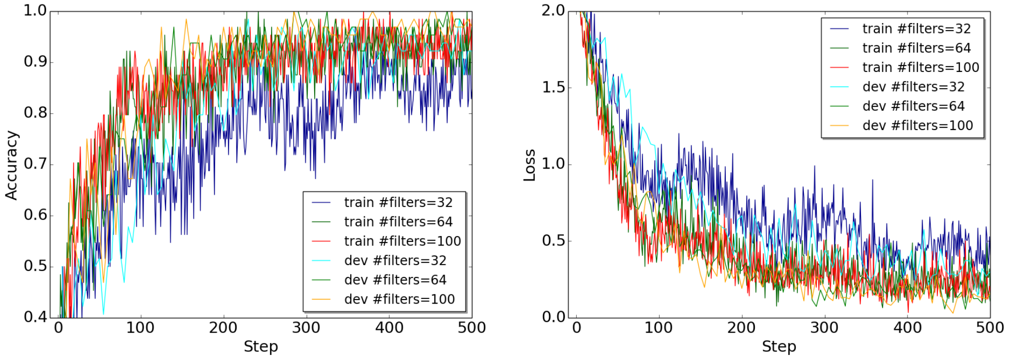


(a) Accuracy Comparison (b) Cross-entropy Comparison

Figure 6.8: Heuristic Search: Embedding size

It can be observed in the plot that both the CNN trained with E=32 and E=64 performed better than the others while performing similar but as a higher E implies a higher training cost it was decided to select an embedding size equals to 32.

3. Number of filters: 32, 64, 100



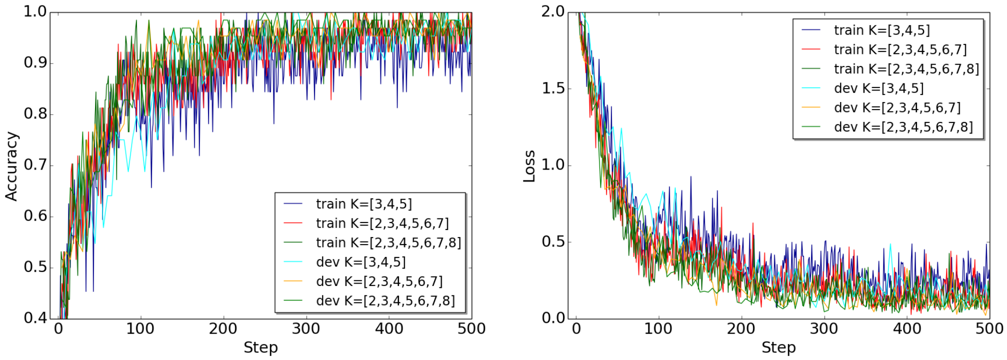
(a) Accuracy Comparison

(b) Cross-entropy Comparison

Figure 6.9: Heuristic Search: #Filters

In this case, the accuracy and the cross-entropy is also practically the same through all the training iterations for #filters=64 and #filters=100 and as the number of filters to learn has a direct relationship with the training cost of the network it was decided to learn 64 filters for each different filter size.

4. Filter sizes: [3,4,5], [2,3,4,5,6,7], [2,3,4,5,6,7,8]



(a) Accuracy Comparison

(b) Cross-entropy Comparison

Figure 6.10: Heuristic Search: Filter Sizes

> Lastly, it was chosen to learn filters with size 2 to size 8 because as it shown in the plot the results obtained are slightly better than the CNN trained with the other set of filter sizes.

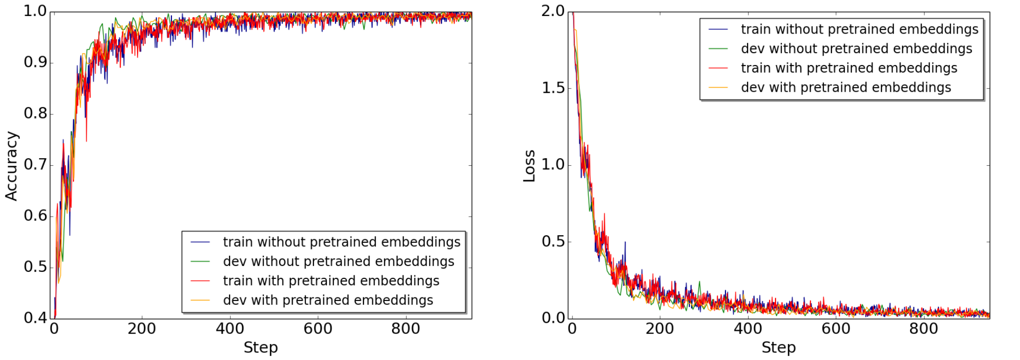
Based on the previous results, the parameters of the network were:

- Learning rate (α) = 0.001
- Embedding size (E) = 32
- #filters = 64
- filter sizes = [2,3,4,5,6,7,8]

After selecting the parameters it was trained the neural network until the validation loss increased in 10 iterations continuously using a mini-batch size equals to 256. However, for comparison purposes, it was decided to limit the number of training epochs to 25. Notice that the number of training iterations per epoch is 39. The number of training iterations per epoch is computed as:

$$\#iterations = N/batch\_size$$

where N is the total size of the training dataset (N=9781). Next you will find their respective confusion matrices and also the accuracy and the cross-entropy of the models until epoch 25.



(a) Training & Validation Accuracy (b) Training & Validation Cross-Entropy

Figure 6.11: Approach B: CNNs training results

- CNN without pretrained word embeddings. Accuracy: 0.9952 Cross-Entropy: 0.0203

	Ramnit	Lollipop	Kelihos_ver3	Vundo	Simda	Tracur	Kelihos_ver1	Obfuscator.ACY	Gatak
Ramnit	1537	0	0	0	0	0	0	4	0
Lollipop	4	2471	0	2	0	0	0	1	0
Kelihos_ver3	0	0	2938	4	0	0	0	0	0
Vundo	0	0	0	474	0	0	0	1	0
Simda	0	0	0	0	42	2	0	0	1
Tracur	3	0	0	1	0	744	0	3	0
Kelihos_ver1	0	0	0	6	0	0	392	0	0
Obfuscator.ACY	11	0	0	9	0	0	0	1208	0
Gatak	0	0	0	0	0	0	0	3	1010

Table 6.1: CNN without pretrained word embeddings: confusion matrix

- CNN with pretrained word embeddings. Accuracy: 0.9947 Cross-Entropy: 0.0243

	Ramnit	Lollipop	Kelihos_ver3	Vundo	Simda	Tracur	Kelihos_ver1	Obfuscator.ACY	Gatak
Ramnit	1528	0	0	8	0	1	0	4	0
Lollipop	2	2473	0	2	0	1	0	0	0
Kelihos_ver3	0	0	2938	4	0	0	0	0	0
Vundo	0	0	0	474	0	0	0	1	0
Simda	0	0	0	0	42	0	0	0	0
Tracur	4	0	0	1	0	746	0	0	0
Kelihos_ver1	0	0	0	6	0	0	392	0	0
Obfuscator.ACY	11	0	0	9	0	1	0	1207	0
Gatak	0	0	0	1	0	2	0	0	1010

Table 6.2: CNN with pretrained word embeddings: confusion matrix

Looking at the confusion matrices it can be observed that this approach do not have the problem of misclassifying samples as belonging to the Gatak family as the approach based on the representation of malware as gray-scale images has. However, it has some difficulty in classifying:

1. samples of the Ramnit family which are incorrectly classified as belonging to the Obfuscator.ACY and viceversa
2. samples of the Tracur family which are misclassified as Ramnit samples
3. samples of the Lollipop family which are incorrectly classified as belonging to the Ramnit or the Vundo family.

Moreover, the CNN with pretrained word embeddings has also misclassified some Ramnit samples as Vundo.

6.3.2 Testing

The CNN models trained during 25 epochs were used to generate the probabilities of each sample from the test set to belong to a malware family. Following you will find the public and the private score achieved by each model.

	Public	Private
without pretrained word embeddings	0.048533931	0.031669778
with pretrained word embeddings	0.048851643	0.036707683

Table 6.3: Approach B: Test scores

The CNN without pretrained word embeddings achieved an improvement of 98,56% with respect to the equal probability benchmark while the CNN with pretrained word embeddings achieved an improvement of 98,33%. Both models are very close in terms of performance and no one can be declared as better than the other. May the fact of generating the vectors represententations of words using samples of malware instead of using goodware is the cause of this situation. Probably if the input of the Skip-Gram model used to learn the word embeddings had been samples of goodware the results would have been quite different. That's because instructions such as xor or nop are commonly used by malware author's for obfuscation purposes. On one hand the exclusive OR (XOR) operation is commonly used to obfuscate particular sensitive strings in the code such as URLs or registry keys. This type of obfuscation works like this:

1. The attacker chooses a 1-byte value to act as a key (range 0 -255).
2. Then malware's code iterates through every byte of the data that needs to be encoded, XOR'ing each byte with the selected key.

When the attacker needs to deobfuscate the string, it repeats the step #2 XOR'ing each byte in the encoded string with the key value.

On the other hand, the No operation (NOP) is used in a technique called dead-code insertion which simply adds some ineffective and redundant instructions to change the appearance of code while keeping its behavior intact.

From my point of view, I think that a better representation of the words in the vocabulary would have been achieved if goodware had been used as the training data because the previous mentioned techniques and lots more not explained in this thesis might produce unnecessary noise in the embeddings.

Chapter 7

Conclusions

This master thesis studies the problem of classifying malware into their corresponding families. In order to explore the problem, we used one of the most recent and biggest datasets publicly available which was provided by Microsoft for the BigData Innovators Gathering Cup (BIG 2015). This dataset provides two files for each malware sample.

1. The hexadecimal representation of the file's binary content.
2. The disassembled file generated by the IDA (Interactive DisAssembler) tool which contains various metadata information extracted from the binary content. Metadata information contains instructions and registers used by the malware, as well as the functions and data imported from DLLs.

This thesis presents two novel and scalable approaches using Convolutional Neural Networks to recognize the family a malware sample belongs.

- The first approach is motivated by [21] which introduced the idea of representing the malware's binary content as gray-scale images. The main benefit of visualizing malware as an image is that the different sections of the binary can be easily differentiated. Their work is based on the observation that images of different malware samples from the same family appear to be similar while images of malware samples belonging to a different family are distinct. This property is useful to classify new malware binaries that have been created by re-using old malware. That's because images are useful to detect small changes while retaining the global structure and the new samples would be very similar visually to the old ones. In consequence, in this thesis, we studied the application of CNNs to learn a feature hierarchy all the way from pixels to the layers of the classifier.

- The second approach uses the architecture introduced by Yoon Kim in [12] for sentence classification but applied to a distinct domain. Instead of classifying sentences from the English language we used their architecture to classify malware samples using the x86 instructions extracted from the disassembled files. We trained two models, the first one without pretrained word embeddings and the second one with pretrained word embeddings generated using the Skip-Gram model.

The first and the second approach obtained a score of 0.134821767 and 0.031669778, respectively. That is an improvement of 93.86% and 98,56% with respect to the equal probability benchmark (logloss=2.1972245577) which is obtained by submitting 1/9 for every prediction. Unfortunately, neither approach outperformed the winner's solution of the competition which obtained a logloss equal to 0.002833228. That's because their solution combined different features such as opcode 2,3 and 4-grams as well as the number of lines per section in the disassembled files, among others. However, the results obtained are quite promising because both approaches are able to classify malware samples much faster than all those solutions that rely on the manually extraction of features and thus, are more scalable.

7.1 Future Work

Even that both approaches have been successfully applied, there is still a huge margin of improvement. From now on, consider that the CNN used to classify malware based on their representation as gray-scale images is named CNN A and the CNN used to classify malware given the x86 instructions of the disassembled file is named CNN B. Hence, one way to improve the results is by merging the output of the last fully-connected layer of CNN A and the output of the max-pooling layer of CNN B. Notice that the resulting CNN should be feed with two types of data: (1) the pixels extracted from the representation of malware as gray-scale images and (2) the opcodes extracted from the disassembled files. The idea is somehow similar to how pretrained word embeddings were load and used in the CNN trained on opcodes. First, both models are trained independently. Therefore, CNN A will learn low-level and high-level features from the images while CNN B will learn a set of features (opcode patterns) from the disassembled files. Then, the learned filters are loaded into the CNN that will combine both approaches and the network is trained again to learn the weight matrix of the output layer. The main advantage of combining both CNNs is that the resulting CNN might be able to solve the problems of each particular approach. In other words, the model may not has problems while classifying Ramnit samples as belonging to the Gatak family and neither discriminating between the Obfuscator.ACY and the Ramnit families.

Additionally, there is still a lot of work to be done regarding the approach presented in section 6. One possible modification would be to increase the number of input opcodes allowed from 10.000 to the maximum number of opcodes in a file found in the entire dataset. By making this small modification, all the information about every malware sample will be captured. In particular, this modification can help to improve the classification rate of samples belonging to the Ramnit and the Lollipop families. That's because samples from both families contain more than 20.000 opcodes per file in average.

Another possible modification is to expand the vocabulary by including the registers, the data directives and the most common imported DLL functions and API calls found in malware. That would be effective for classifying all those samples that, because of applying code obfuscation techniques, do not have any instruction. In contrast, by expanding the vocabulary, the number of parameters to learn will increase, as well as the time needed for training the CNN.

Finally, the word embeddings could be improved by using as training data samples of executables belonging to goodware. The main idea is that there are some of the disassembled files that have been obfuscated by applying different techniques such as using the dead-code insertion technique or the XOR operation, among others. In consequence, the word embeddings may not be accurate enough and might not represent correctly the words in the vocabulary.

Bibliography

[1] Mansour Ahmadi, Dmitry Ulyanov, Stanislav Semenov, Mikhail Trofimov, and Giorgio Giacinto. Novel feature extraction, selection and fusion for effective malware family classification. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, CODASPY '16, pages 183–194, New York, NY, USA, 2016. ACM.

[2] Clint Feher Shlomi Dolev Asaf Shabtai, Robert Moskovitch and Yuval Elovici. Detecting unknown malicious code by applying classification techniques on opcode patterns. In *Security Informatics*. 2012.

[3] Daniel Billar. Opcodes as predictor for malware. *International Journal of Electronic Security and Digital Forensics*, 1:156–168, 2007.

[4] R Manoharan Chandrasekar Ravi. Malware detection using windows api sequence and machine learning. *International Journal of Computer Applications*, 43, 2012.

[5] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12:2121–2159, July 2011.

[6] Ekta Gandotra, Divya Bansal, and Sanjeev Sofat. Malware analysis and classification: A survey. *Journal of Information Security*, pages 56–64, 2014.

[7] Dragos Gavrilut, Mihai Cimpoes, Dan Anton, and Liviu Ciortuz. Malware detection using machine learning. *Proceedings of the International Multiconference on Computer Science and Information Technology*, page 735–741, 2009.

[8] Li Deng George E. Dahl, Jack W. Stokes and Dong Yu. Large-scale malware classification using random projections and neural network. *ICASSP*, 2013.

[9] Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines vinod nair. In *Proceedings of the 27th International Conference on Machine Learning (ICML),*, pages 807 – 814, 2010.

[10] S. Hochreiter. Untersuchungen zu dynamischen neuronalen netzen. diploma thesis, institut fur informatik, lehrstuhl prof. brauer, technische universitat munchen, 1991.

[11] Javier Nieves Yoseba K. Penya Borja Sanz Igor Santos, Felix Brezo and Carlos Laorden. Opcode-sequence-based malware detection. In *Engineering Secure Software and Systems*, volume 5965.

[12] Yoon Kim. Convolutional neural networks for sentence classification. *CoRR*, abs/1408.5882, 2014.

[13] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[14] J.Z. Kolter and M.A. Maloof. Learning to detect and classify malicious executables in the wild. *Journal of Machine Learning Research*, page 2721–2744, 2006.

[15] Deguang Kong and Guanhua Yan. Discriminant malware distance learning on structural information for automated malware classification. Technical report, SIGMETRICS'13, 2013.

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, 2012.

[17] Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J. Smola. Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 661–670, New York, NY, USA, 2014. ACM.

[18] Robert Lyda and James Hamrock. Using entropy analysis to find encrypted and packed malware. *IEEE Security and Analysis*, 5:40–45, 2007.

[19] Zahra Salehi Mahboobe Ghiasi, Ashkan Sami. Dynamic malware detection using registers values set analysis.

[20] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 3111–3119. Curran Associates, Inc., 2013.

[21] L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath. Malware images: Visualization and automatic classification, 2011.

[22] Anil Thomas Nikos Karampatziakis, Jack Stokes and Mady Marinescu. Using file relationships in malware classification. *Detection of Intrusions and Malware, and Vulnerability Assessment*, 7591:1–20, 2013.

[23] Aude Oliva and Antonio Torralba. Modeling the shape of the scene: A holistic representation of the spatial envelope. *International Journal of Computer Vision*, 42:145–175, 2001.

[24] Aude Oliva and Antonio Torralba. Modeling the shape of the scene: A holistic representation of the spatial envelope. *Int. J. Comput. Vision*, 42(3):145–175, May 2001.

[25] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural Netw.*, 12(1):145–151, January 1999.

[26] Smita Ranvee and Swapnaja Hiray. Comparative analysis of feature extraction methods of malware detection. *International Journal of Computer Applications*, 120, 2015.

[27] Clint Feher Nir Nissim Robert Moskovitch, Dima Stopel and Yuval Elovici. Unknown malcode detection via text categorization and the imbalance problem. *IEEE International Conference on Intelligence and Security Informatics*, pages 156–161, 2008.

[28] Joshua Saxe and Konstantin Berlin. Deep neural network based malware detection using two dimensional binary program features, 2015.

[29] Sjsu Scholarworks and Donabelle Bays. Structural entropy and metamorphic malware, 2013.

[30] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, SP '01, pages 144–, Washington, DC, USA, 2001. IEEE Computer Society.

[31] Ohm Sornil and Chatchai Liangboonprakong. Malware classification using n-grams sequential pattern feature. *International Journal of Information Processing and Management*, 4, 2013.

[32] 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.

[33] Kephart J.O. Tesauro, G.J. and Gregory B Sorkin. Neural networks for computer virus recognition. *IEEE International Conference on Intelligence and Security Informatics*, 11, 1996.

[34] Antonio Torralba, Kevin P. Murphy, William T. Freeman, and Mark Rubin. Context-based vision system for place and object recognition. pages 273–280, 2003.

[35] L.J.P. van der Maaten and G.E. Hinton. Visualizing high-dimensional data using t-sne. 2008.

[36] Nitin Rai Veeramani R. Windows api based malware detection and framework analysis. *International Journal of Scientific & Engineering Research*, 3, 2012.

[37] Yanfang Ye, Lifei Chen, Dingding Wang, Tao Li, Qingshan Jiang, and Min Zhao. Sbmds: an interpretable string based malware detection system using svm ensemble with bagging, 2009.