ASSIGNMENT OF BACHELOR’S THESIS

Title:	Static malware detection using recurrent neural networks
Student:	Matouš Kozák
Supervisor:	Mgr. Martin Jureček
Study Programme:	Informatics
Study Branch:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2020/21

Instructions

The goal of the thesis is an application of various types of recurrent neural networks (RNN) especially Long Short-term Memory (LSTM) to static malware detection.

Instructions:

1) study and describe the concept of RNN, focusing on LSTM
2) collect benign and download malware samples (e.g from open repository virushare.com)
3) transform the binaries to feature vectors using static analysis and perform common feature selection and normalization methods
4) use existing machine learning libraries (Keras or TensorFlow, ...) and apply various architectures and topologies of RNN to malware detection
5) discuss the results and compare them with results of other supervised learning algorithms
6) try to find the most appropriate structure of LSTM for the given problem of malware detection

References

Will be provided by the supervisor.

doc. Ing. Jan Janoušek, Ph.D. Head of Department	doc. RNDr. Ing. Marcel Jiřina, Ph.D. Dean

Prague December 19, 2019

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Static Malware Detection using Recurrent Neural Networks

Matouš Kozák

Department of Theoretical Computer Science
Supervisor: Mgr. Martin Jureček

June 2, 2020

Acknowledgements

Firstly, I would like to thank my supervisor Mrg. Martin Jureček for his introduction to malware detection research and valuable guidance in this topic. Next, I wish to thank my parents for never-ending support in my life, not only in my pursuit of education.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on June 2, 2020 . . . .

Czech Technical University in Prague
Faculty of Information Technology
© 2020 Matouš Kozák. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Kozák, Matouš. *Static Malware Detection using Recurrent Neural Networks.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstrakt

Neustále rostoucí počty útoků škodlivých programů na naši IT infrastrukturu si žádají nové a lepší metody ochrany. V této bakalářské práci se věnujeme využití rekurentních neuronových sítí pro rychlou a přesnou detekci malwaru. Pro reprezentaci podezřelých programů jsme využili pouze data extrahovaná ze souborů v PE formátu. Tato data jsme dále použili pro trénink různých druhů rekurentních neuronových sítí. V práci představujeme speciální architekturu neuronové sítě, kombinující husté a LSTM vrstvy, ke klasifikaci PE souborů. Pracovali jsme s naším vlastním datasetem obsahující 30154 souborů stažených z dostupných zdrojů. S tímto datasetem, který je rovnoměrně rozdělen mezi čisté a škodlivé soubory, jsme dosáhli přesnosti 98,41 % s pouze 0,5 % legitimních programů mylně klasifikovaných jako malware. K těmto výsledkům nám stačilo pouhých 250 iterací přes treninkový soubor vzorků k naučení naší sítě. Výsledky dokazují, že algoritmy strojového učení, hlavně LSTM sítě, mohou být využity jako rychlý a spolehlivý nástroj pro detekci škodlivých souborů.

Klíčová slova detekce malware, rekurentní neuronové sítě, LSTM, statická analýza, PE soubory, strojové učení

Abstract

An ever-growing number of malicious attacks on our IT infrastructure calls for new and better methods of protection. In this thesis, we focus on the use of recurrent neural networks as an agile and accurate way of detecting malware. We only used features extracted from files in the PE file format to represent the suspicious programs which we used to train various types of recurrent neural networks. In this work, we present unique neural network architecture combining dense and stacked LSTM layers to classify PE files. We worked with our dataset of 30,154 files collected from available resources with which we achieved an accuracy of 98.41%, while only 0.5% of benign samples were misclassified as malware on our balanced dataset. All this was accomplished with only 250 epochs of training. These results prove that machine-learning algorithms, especially LSTM networks, can be used as a quick and reliable tool for malware detection.

Keywords malware detection, recurrent neural networks, LSTM, static analysis, PE files, machine learning

Contents

Introduction **1**

1 Neural Networks **3**

1.1 Artificial Neuron 3
1.1.1 Perceptron 3
1.1.2 Sigmoid Neuron 4
1.1.3 Activation Function 5
1.1.3.1 Sigmoid Function 5
1.1.3.2 Hyperbolic Tangent 6
1.1.3.3 Rectified Linear Unit 6
1.1.3.4 Softmax 7
1.2 Types of Neural Networks 7
1.2.1 Feed-Forward Networks 8
1.2.1.1 Backpropagation 8
1.2.2 Convolutional Neural Networks 9
1.2.3 Recurrent Neural Networks 10
1.2.3.1 Bidirectional Recurrent Neural Networks . . . 11
1.2.4 Long Short-Term Memory 12
1.2.4.1 Bidirectional Long Short-Term Memory 14
1.2.5 Gated Recurrent Unit 15
1.2.6 Autoencoder 15

2 Portable Executable **17**

2.1 Structure of PE File 17
2.1.1 MS-DOS Header and Stub 17
2.1.2 File Header 18
2.1.3 Optional Header 18
2.1.4 Section Headers 18
2.1.5 Section Data 18

3 Experiments **19**
3.1 Dataset Description . . . 19
3.1.1 Our Dataset . . . 19
3.1.2 EMBER Malware Dataset . . . 19
3.2 Feature Extraction . . . 20
3.3 Feature Preparation . . . 20
3.3.1 Vectorization . . . 21
3.3.2 Hashing . . . 21
3.4 Feature Selection . . . 22
3.5 Proposed Method . . . 23
3.5.1 Approximate Testing . . . 23
3.5.2 Promising Structures . . . 24

4 Results **27**
4.1 Evaluation Metrics . . . 27
4.1.1 Confusion Matrix . . . 27
4.1.2 ROC Curve . . . 29
4.2 Supervised ML Algorithms . . . 29
4.3 Comparison with the State-of-the-art . . . 32

Conclusion **33**

Bibliography **35**

A Acronyms **41**

B Contents of enclosed CD **43**

List of Figures

1.1 Perceptron 4
1.2 Comparison of the sigmoid and step function 5
1.3 Comparison of the hyperbolic tangent with sigmoid function . . . 6
1.4 Rectified Linear Unit 7
1.5 Fully connected feed-forward neural network 8
1.6 Convolution matrix 9
1.7 Unrolled Recurrent Neural Network 10
1.8 Unrolled version of BRNN 11
1.9 Example of LSTM architecture 13
1.10 Structure of BLSTM network 14
1.11 Example of autoencoder for digit compression [26] 16

2.1 Structure of a PE file 17

3.1 ROC curve of different feature selection techniques 23
3.2 Proposed structure of LSTM network 25
3.3 Evolution of ACC, TPR and FPR over 700 epochs on the validation set 26
3.4 Confusion matrix presenting our results 26

4.1 Confusion matrix, 0 stands for negative and 1 for positive values . 28
4.2 ROC curve of tested ML algorithms 30
4.3 ROC curve of tested ML algorithms (EMBER) 31

List of Tables

3.1 Comparison of different selection techniques 22
3.2 The most promising RNN architectures 24
3.3 Top five RNN architectures after 200 epochs 25

4.1 Comparison of our LSTM network with well-known supervised machine-learning algorithms 30
4.2 Comparison of our LSTM network with well-known supervised machine-learning algorithms (EMBER) 31

Introduction

Malware is a software which conducts malicious activities on the infected computer. Cybersecurity professionals across the globe are trying to tackle this unwanted behaviour. Even though they are developing defence systems on a daily basis, cybercriminals process at the same, if not, faster manner.

Antivirus programs detect more than 370,000 malicious programs each day [1], and the number keeps rising. Although Windows remains the most attacked platform, macOS and IoT devices are becoming attractive targets as well. The most popular weapon for cybercriminals on Windows remains Trojan, for instance, Emotet, WannaCry, Mirai and many others [2].

In May 2017, the world was struck by new ransomware (type of malware) WannaCry. This virus quickly spread all around the world, infecting more than 230,000 computers in 150 countries. Between infected organizations were, e.g. FedEx, O2 or Britain's NHS and the cost of damage was estimated at around 4 billion dollars [3].

Long-established malware detection methods use signatures to identify malicious software [4]. This approach is highly accurate on already known malware files. The signatures can be easily generated and compared with the database. However, they can be easily evaded by encrypting or obfuscating the program. Another disadvantage of this technique is the encounter with unknown files which could go undetected for some time until they're recognized as malware, and their signature is added to the database. This delay could be crucial as malware evolves fast and zero-day attacks are as dangerous as any.

The ambition of a machine learning approach to this issue is to overcome this weakness and provide fast and accurate malware detection.

In the thesis, we focus on static analysis for multiple reasons. Firstly, extracting API calls from executable files needs to be performed in a sandbox environment to secure leak of possible malicious activities into our system. However, this is bypassed by unnatural behaviour of many programs in these surroundings. Secondly, it's time-consuming running large datasets and cap-

turing their activities. These are the reasons why we focused on static malware analysis.

The goal of this thesis is to found the most appropriate structure of long short-term memory (LSTM) network for malware detection. This problem will be tackled in two stages. The first stage will be collecting malware and benign files, extracting useful information, transforming and selecting the best features to create our dataset. The second stage will consist of testing different recurrent neural network architectures and evaluating our results. We will perform static analysis, that means only working with information available from gathered files without studying their working behaviour.

Structure of the thesis is as follows:

In Chapter 1, we introduce neural networks, explain basic terms and simple net architectures. Further, we describe various advanced structures used in cutting edge applications.

In Chapter 2, we briefly explain the PE file format. Where it's used and what structure it has.

In Chapter 3, we present the tools and techniques we used. From feature engineering and selection to presenting our approaches to static malware detection.

In Chapter 4, we describe metrics used for evaluating our work, demonstrate our results and compare them with related work.

In Conclusion, we will evaluate the results of the work and suggest possibilities for future research.

Chapter **1**

Neural Networks

Neural network (**NN**) is a mathematical model inspired by human brains. As described in [5], the network consists of units (neurons) and connections (synapses). Typically, units are divided into many layers, starting with the *input layer*, followed by several *hidden layers* and terminated with the *output layer*.

Even though the first research about NN dates back to 40s when McCulloch and Pitts published in 1943, *A logical calculus of the ideas immanent in nervous activity* [6], they didn't gain much spotlight until recent years. Thanks to rapidly increasing available computational power, training data and new models, neural networks are becoming the go-to model for solving complex tasks which we thought to be unsolvable by computers [7].

In this chapter, we are going to look at different types of neural units and activation functions they use. Then we will briefly look at backpropagation algorithm and go through a variety of neural network architectures, starting with feed-forward and convolutional neural networks, finishing with different recurrent neural nets, for instance, long short-term memory architecture.

1.1 Artificial Neuron

Artificial neurons are units which try to imitate the behaviour of real brain neurons. Same as brain neurons, they connect with other neurons and pass information between themselves. The simplest type of artificial neuron is *perceptron* [8].

1.1.1 Perceptron

We can imagine **perceptron** as a node with a couple of incoming edges and one outgoing edge [9]. Incoming edges represent a binary input, vector $\vec{x} = (x_1, x_2, \ldots, x_n)$. To differentiate significance between input edges we use real-valued numbers assigned to each edge, vector of weights $\vec{w} = (w_1, w_2, \ldots, w_n)$.

The node also contains the parameter *threshold*, a real number as well. The output is a binary value, 0 or 1.

To determine the output value, we must at first calculate the weighted sum $\alpha = \sum_i x_i w_i$ and then check if the value exceeds the *threshold* value.

$$output = \begin{cases} 1, & \text{if } \alpha \geq threshold \\ 0, & \text{if } \alpha < threshold \end{cases} \tag{1.1}$$

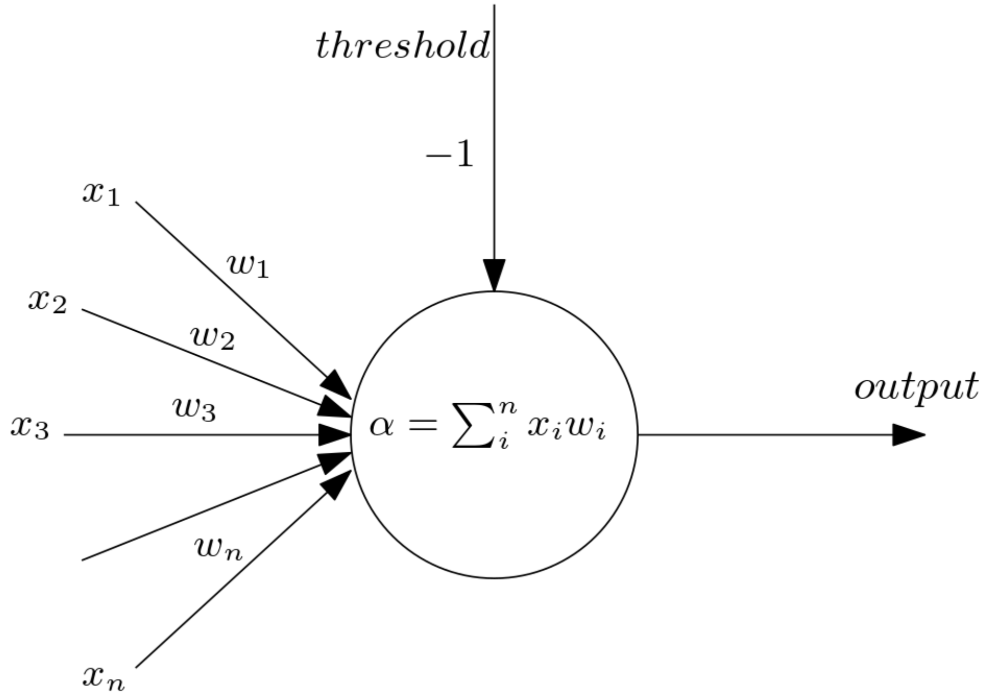


Figure 1.1: Perceptron

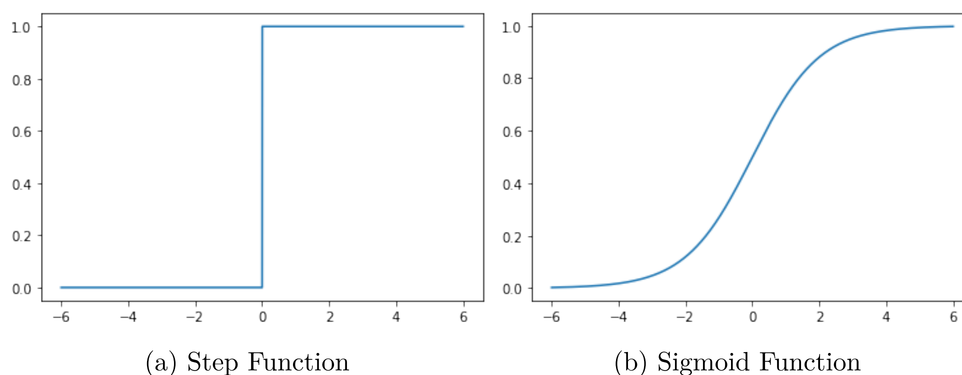
Often we represent *threshold* as an additional incoming edge with negative weight −1, as shown in Figure 1.1.

This Figure 1.1 and subsequent illustrations were created using the `Ipe` drawing editor [10].

1.1.2 Sigmoid Neuron

A more popular type of artificial neuron is called a **sigmoid neuron** [8]. Just like perceptron, the sigmoid neuron has inputs $\vec{x}$ and weights $\vec{w}$. The difference comes in output value and how we calculate it. Sigmoid neuron doesn't have binary output but a real-valued number between 0 and 1. To compute output value, we use a sigmoid function.

$$\sigma(\alpha) = \frac{1}{1 + e^{-\alpha}} \tag{1.2}$$



(a) Step Function (b) Sigmoid Function

Figure 1.2: Comparison of the sigmoid and step function

In the Figure 1.2 we can see that sigmoid function (1.2b) is a smoothed out version of step function (1.2a).

The sigmoid function is just one type of *activation function* used in artificial neurons [9]. In perceptron 1.1, we used the so-called step function. More commonly used types include *tanh*, *ReLU* or *softmax*.

1.1.3 Activation Function

Activation function inside the artificial neuron is used to define the node's output, typically in the form of $f : \mathbb{R} \to \mathbb{R}$ [11]. Important trait of many activation functions is its differentiability. Reason for that is the *Backpropagation algorithm* used for learning the weights of NN. The additional necessary feature is, the derivative of the activation function doesn't *saturate* nor *explode*, heads towards 0 or ∞. These are the reasons why perceptrons with step function aren't suitable to be used in the present models [12].

1.1.3.1 Sigmoid Function

Already mentioned in 1.1.2 and described by Leskovec et al. in [11], the **sigmoid function** is commonly used in artificial neurons as an alternative to step function. Most used choice of the sigmoid function is **logistic sigmoid**:

$$\sigma(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{1 + e^x} \tag{1.3}$$

One of the reasons logistic sigmoid is so popular is the result of it's derivative.

$$\frac{d}{dx}\sigma(x) = \frac{e^x(1 + e^x) - e^x e^x}{(1 + e^x)^2} = \frac{e^x}{(1 + e^x)^2} = \sigma(x)(1 - \sigma(x)) \tag{1.4}$$

One complication logistic sigmoid has, its derivative quickly saturates when we move outwards from the area around 0. This problem hinders the learning process of NN.

1.1.3.2 Hyperbolic Tangent

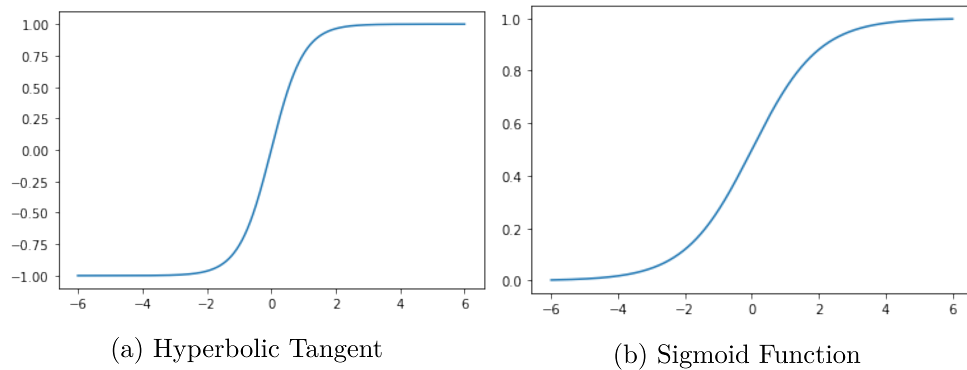
As well as a logistic sigmoid, **hyperbolic tangent** (**tanh**), is a version of the sigmoid function:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{1.5}$$

It's output isn't constrained in range 0 to 1, but between -1 and 1. Same as with logistic sigmoid, the derivation of tanh can be calculated simply:

$$\begin{aligned} \frac{d}{dx} tanh(x) &= \frac{(e^x + e^{-x})(e^x + e^{-x}) - (e^x - e^{-x})(e^x - e^{-x})}{(e^x + e^{-x})^2} \\ &= 1 - \frac{(e^x - e^{-x})^2}{(e^x + e^{-x})^2} = 1 - tanh^2(x) \end{aligned} \tag{1.6}$$

Since it is only moved and scaled version of logistic sigmoid, it shares its problems with derivative saturation [11]. The contrast between these two function can be seen in Figure 1.3. Pay attention to the scale of the y-axis.



(a) Hyperbolic Tangent

(b) Sigmoid Function

Figure 1.3: Comparison of the hyperbolic tangent with sigmoid function

1.1.3.3 Rectified Linear Unit

Different type of activation function is **rectified linear unit** (**ReLU**). The output of the rectified linear unit is defined as:

$$f(x) = max(0, x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases} \tag{1.7}$$

ReLU became very popular thanks to its ease of function and derivative computation [11]. Also, for the positive x, the function's gradient remains constant.

$$\frac{d}{dx}max(0,x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{if } x < 0 \end{cases} \tag{1.8}$$

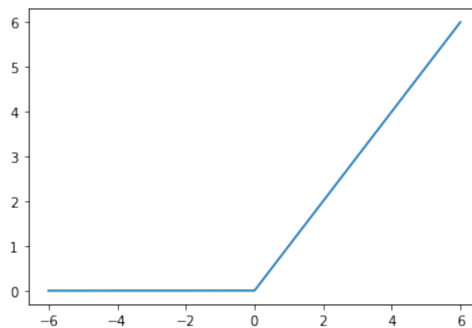


Figure 1.4: Rectified Linear Unit

Even though the ReLU isn't differentiable at 0, it is everywhere else. Another problem the function has is saturation for negative x. This might cause a so-called *dying ReLU problem*. Today, there are many variations of ReLU, e.g. *ELU*, *Leaky ReLU*, *SELU*, ...

1.1.3.4 Softmax

All the previous activation functions operated on a single input value. **Softmax** on the other hand, works with vectors. Input to the softmax function is a vector $\vec{x} = (x_1, x_2, \ldots, x_n)$ and output for each x_i, $i \in 1, \ldots, n$ is defined as:

$$\sigma(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \tag{1.9}$$

We can see that $\sum_i \sigma(x_i) = 1$. Thus it can be interpreted as a probability distribution as explained in [12]. For this characteristic, it's being used in multiclass classification task inside the output layer for dividing results into k probability groups.

1.2 Types of Neural Networks

Nowadays, there are many varieties of neural networks, each with its own structure and use cases. In this part, we present the most common types, such as *feed-forward* or *recurrent* neural nets.

1.2.1 Feed-Forward Networks

Feed-forward network (**FFN**) is named by the flow of data inside them. All edges are oriented “forward”, from the *input layer* to *output layer*, without cycles. They may contain a number of *hidden layers* with various widths. The width and number of hidden layers are pivotal in designing FNNs, but there isn’t a general rule to help you decide [11]. An example of FNN can be seen in Figure 1.5 below.

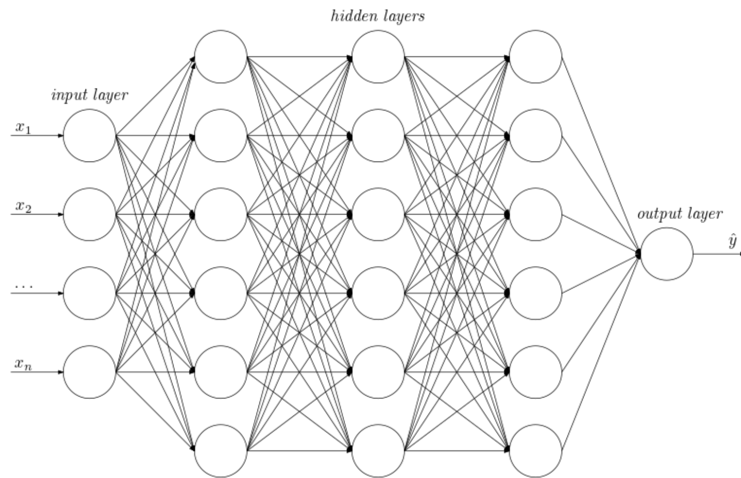


Figure 1.5: Fully connected feed-forward neural network

The input layer receives input, vector $\vec{x}$, and the output layer produces output $\hat{y}$. The process of learning the weights of the network consists of minimizing the *loss function* $\mathcal{L}(\hat{y}, y)$, where y is target output for input $\vec{x}$ [12].

1.2.1.1 Backpropagation

Backpropagation, shorting of *backward propagation of errors* is a common algorithm used for learning the NN [11]. It uses gradient descent to update the weights:

$$\vec{w}_{i+1} \leftarrow \vec{w}_i - \gamma \nabla F(\vec{w}_i) \tag{1.10}$$

Here, $\vec{w}$ are the weights (the parameters) of the network, γ is the learning rate and $\nabla F(\vec{w}_i)$ is the gradient value at current state i. The goal of gradient descent is to get closer to the minimum of *loss function* $\mathcal{L}$.

The algorithm terminates either when the change between the following iterations becomes small enough or reaches the maximum number of repetitions. Both conditions are heavily influenced by the size of the learning rate γ. Since optimization of FNNs is NP-hard problem as proven by Blum and Rivest in [13], there is no guarantee that this method hits the global minimum,

as the loss function's surface doesn't have to be convex, it might end up in local minimum from which it can't escape. However, there had been much work on optimization techniques to achieve satisfactory results [12].

1.2.2 Convolutional Neural Networks

Specially designed type of FNN is a **convolutional neural network** (**CNN**). These networks were found highly effective in processing images and other two-dimensional inputs [5]. CNNs usually consist of several *convolutional* layers interleaved with standard *pooling* layers.

In a *convolutional* layer, units are ordered in a two-dimensional grid. The node at position (x, y) receives inputs from units in the previous layer at positions (x_i, y_j), where $x \leq x_i \leq x + k$, $y \leq y_j \leq y + l$ and $k,\ l$ are constants for a given layer. Rather than having different weights for each connection from the previous layer, we use a single matrix $W_{k,l}$. The node's output is then calculated in the manner as with regular layers. In the first place, we calculate the weighted sum of all inputs' values with weights $W_{k,l}$ and then feed the result into any activation function.

1	0	0
0	0	0
0	0	0

Figure 1.6: Convolution matrix

By using single matrix $W_{k,l}$ for given k and l, we radically decrease the number of parameters network needs to learn and as a result which increases the training speed [9]. Convolution matrix $W_{3,3}$ representing identity is shown in Figure 1.6.

The idea behind using convolutions came from the human eye, where neurons are arranged in layers and only connected to close surroundings. This arrangement is being mimicked in today's CNNs. Same as in human eye, the sequence of convolutional layers attempts to detect various shapes, from simple lines, up to more complex structures, such as bicycle wheel or human face [11].

Typical components of CNNs are *pooling* layers. The pooling layer operates as a function which takes as an input area of small size and returns the single output. An example of pooling function might be *max-pooling* that returns the maximum value from a given area. Other functions can be used, such as average. Note that pooling layers reduce the dimension by summarizing the

input. The intention behind this simplification is to make output invariant to translations and other picture modifications [5].

1.2.3 Recurrent Neural Networks

Different kind of NN is a **recurrent neural network** (**RNN**). Unlike FNN, this structure allows cycles. There isn't a single input layer and output layer, but input is fed and the output produced in each layer of RNN. Every node has a self-recurring edge that passes information from the past and then to the future. This sequential process was found exceptionally successful in processing time, language or video data [11].

The input to RNN is a vector $\vec{x} = (x_1, x_2, \ldots)$ and output of the network is also a vector $\vec{\hat{y}} = (\hat{y}_1, \hat{y}_2, \ldots)$. We can imagine every layer of RNN as a vector of nodes at time t, where all units at time t collect input x_t and value from the previous *hidden state* h_{t-1}. The **hidden state** is a vector which serves as current memory we are working with and is calculated as follows:

$$h_t = f(W_x x_t + W_h h_{t-1} + \vec{b_h}) \tag{1.11}$$

Where f is the activation function, W_x, W_h are learned matrices of weights and $\vec{b}_h$ is a vector of bias parameters learned as well. Hidden state h_0 is set to $(0, 0 \ldots, 0)$. The output $\hat{y}_t$ is then calculated as:

$$\hat{y}_t = g(W_y h_t + \vec{b}_y) \tag{1.12}$$

Here g is also an activation function. Usually, we use softmax to make sure the output is in the desired class range. W_y is a matrix of weights and $\vec{b}_y$ a vector of biases determined during the learning process.

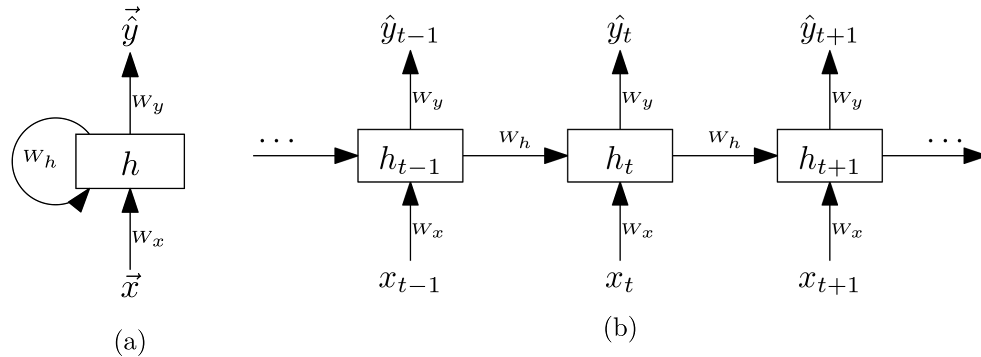


Figure 1.7: Unrolled Recurrent Neural Network

For learning a RNN, we use a modified version of the backpropagation algorithm, **backpropagation through time** (**BPTT**). It works by unrolling the RNN [5], calculating the losses across time steps. Then with the help of the backpropagation algorithm updating the weights. The unrolled recurrent

neural network is shown in the Figure 1.7b above. Detailed description of RNNs can be found in [12] by Lipton et al.

One of the significant weaknesses RNNs have is the problem of *vanishing gradient*. This problem occurs when we try to detect long-distance dependencies, e.g. subject-verb connexion in long sentences. Since the information in RNNs is only passed between adjacent nodes, there is a minimal contribution from distant layers [11].

1.2.3.1 Bidirectional Recurrent Neural Networks

For some tasks, the standard RNNs aren't suitable as the hidden state is determined only by prior states. Such tasks include text and speech recognition or others where the output at time t depends on the past as well as future inputs or labelling problems where the output is expected after finishing processing the whole input sequence [14]. **Bidirectional Recurrent Neural Networks** (**BRNNs**) try to solve this problem by introducing second backwards hidden state for each time t, both connected to the output. Input to the BRNN is then presented in two rounds, once forwards as with RNN and then in a reversed direction from the back. The two hidden states are then combined to compute the output $\hat{y}_t$. This architecture was presented by Schuster et al. in [15].

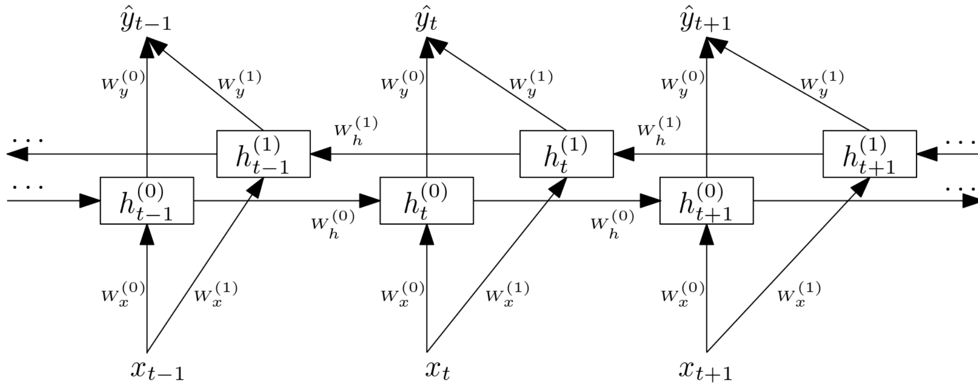


Figure 1.8: Unrolled version of BRNN

In Figure 1.8, the (0) and (1) in superscripts denote forwards and backwards flows respectively. Remark that the weight matrices are also distinct for both hidden states as described in [12].

The hidden states are updated identically as with RNN.

$$h_t^{(0)} = f(W_x^{(0)} x_t + W_h^{(0)} h_{t-1}^{(0)} + \vec{b}_{h^{(0)}}) \tag{1.13}$$

$$h_t^{(1)} = f(W_x^{(1)} x_t + W_h^{(1)} h_{t-1}^{(1)} + \vec{b}_{h^{(1)}}) \tag{1.14}$$

The output $\hat{y}_t$ is then calculated with the use of both hidden states:

$$\hat{y}_t = g(W_y^{(0)} h_t^{(0)} + W_y^{(1)} h_t^{(1)} + \vec{b}_y) \tag{1.15}$$

All the activation functions and parameters are the same as with vanilla RNN. This model has its drawbacks as it needs a fixed length of the net and introduced second hidden state increases the number of parameters needed to be learned [12]. However, for many problems consisting of limited input size, it achieves better results than plain RNN [14].

1.2.4 Long Short-Term Memory

Improved version of RNN solving the problem of vanishing gradient is **Long short-term memory network** or shortly **LSTM**. This network architecture was introduced in [16] by Hochreiter and Schmidhuber in 1997. The improvement lies in replacing a simple node from RNN with a compound unit consisting of hidden state (as with RNNs) and so-called *cell state* or c_t. Further, adding *input node* g_t compiling the input for every time step t and three *gates* controlling the flow of information. Gates are binary vectors, where 1 allows data to pass through, 0 blocks the circulation and operations with gates are handled by using Hadamard (element-wise) product $\odot$ with another vector [11].

As mentioned above, LSTM cell is formed by a group of simple units. The key difference from RNN is the addition of three **gates** which regulate the input/output of the cell.

Note that W_x, W_h and $\vec{b}$ with subscripts in all of the equations below are learned weights matrices and vectors respectively, and f denotes an activation function, e.g. sigmoid. Subscripts are used to distinguish matrices and vectors used in specific equations.

1. **Input gate**: Determines which information can be allowed inside the unit:

$$i_t = f(W_{x_i} x_t + W_{h_i} h_{t-1} + \vec{b}_i) \tag{1.16}$$

2. **Forget gate**: Allows us to discard information from memory we don't longer need:

$$f_t = f(W_{x_f} x_t + W_{h_f} h_{t-1} + \vec{b}_f) \tag{1.17}$$

3. **Output gate**: This gate learns what data is paramount at a given moment and enables the unit to focus on it:

$$o_t = f(W_{x_o} x_t + W_{h_o} h_{t-1} + \vec{b}_o) \tag{1.18}$$

The **input node** takes as an input x_t and previous hidden state:

$$g_t = f(W_{x_g} x_t + W_{h_g} h_{t-1} + \vec{b}_g) \tag{1.19}$$

As an activation function is typically used tanh even though ReLU might be easier to train [12].

The **cell state** is calculated as follows:

$$c_t = i_t \odot g_t + f_t \odot c_{t-1} \tag{1.20}$$

In equation 1.20, we can see the intuition behind using the input and forget gates. The gates handle how much of the input node and previous cell state we allow into the cell. This formula is the essential improvement to simple RNNs as the forget gate vector applied to the previous cell state is what allows the gradient to safely pass during backpropagation, thus abolishing the problem of *vanishing gradient* [11].

The **hidden state** is then updated with the content of the current cell state modified with output gate o_t.

$$h_t = f(c_t \odot o_t) \tag{1.21}$$

Here, the activation function f is usually a *tanh* function. We can imagine the hidden state is the short-term memory and cell state the long-term memory of LSTM network.

The output $\hat{y}_t$ is then computed in the same way as with RNN (1.12).

$$\hat{y}_t = f(W_{h_y} h_t + \vec{b}_y) \tag{1.22}$$

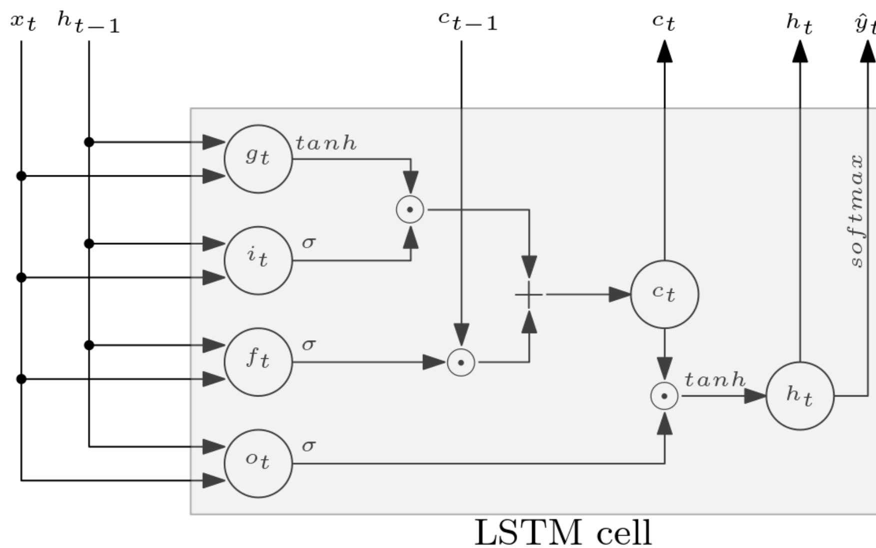


Figure 1.9: Example of LSTM architecture

Presented LSTM architecture closely maps the state-of-the-art design from [17]. Note that we dropped the network's parameters, matrices of weights and vectors of biases to keep it well-arranged.

1.2.4.1 Bidirectional Long Short-Term Memory

Same as with RNN and BRNN we can imagine LSTM net having connections both from the past and future cells. This architecture is called **Bidirectional Long Short-Term Memory** (**BLSTM**) and was introduced by Graves and Schmidhuber in 2005 [18].

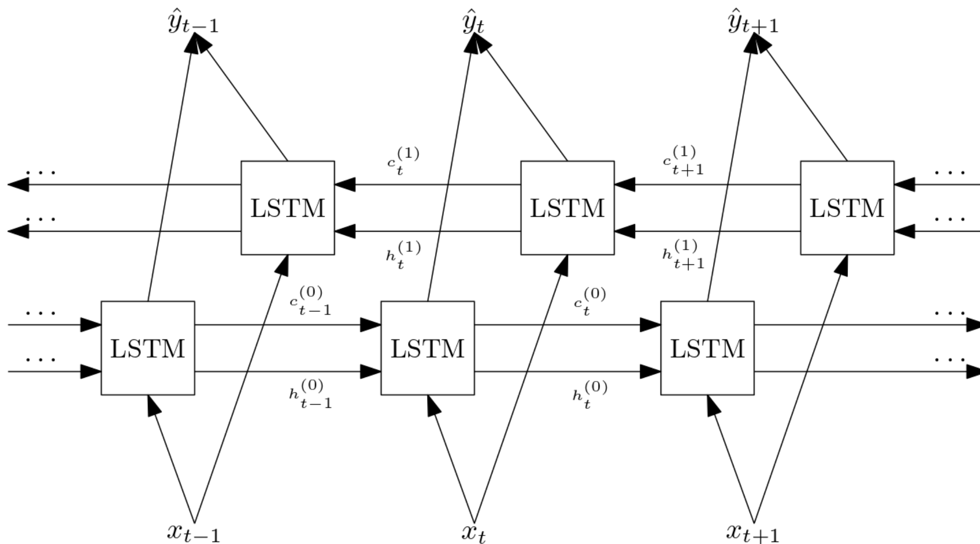


Figure 1.10: Structure of BLSTM network

In Figure 1.10 we can see structure of BLSTM, the (0) and (1) in superscripts stand for forwards and backwards directions, respectively. We omitted detailed representation of LSTM cells to make the illustration simpler. To see the detailed illustration of LSTM cell see Figure 1.9.

BLSTMs can be used to solve similar problems as bidirectional RNNs where we have entire input available beforehand. Training the network in a forwards and backwards directions helps to gain context from past and future as well [19]. In addition, having hidden and cell state enables better storage of information across the timeline even from distant past or future.

Bidirectional LSTMs were found to outperform standard BRNNs in many tasks, e.g. speech recognition. This was proven in the first application of BLSTMs by Graves et al. for phoneme classification problem [18]. Later, they were also used for handwriting recognition where they achieved state-of-the-art results as well [20, 21]. In [22], BLSTM was used as a parser for effective feature representation. Sundermeyer et al. used BLSTMs for lan-

guage translation and scored good results [23]. Further, they were also used in protein structure prediction as described in [24].

As with other bidirectional RNNs, BLSTMs aren't suitable for problems where we don't know the final length of the input and the results are required after each timestamp (*online* tasks).

1.2.5 Gated Recurrent Unit

Popular modification of LSTM is **Gated recurrent unit** (**GRU**). It uses only one *hidden* state instead of two as found in LSTM. As a result, it contains fewer parameters and may be easier to train than LSTM [11].

As a consequence of not having a separate cell and hidden state, it uses one less gate. The two gates are called *update* and *reset* gate and are calculated as follows:

1. **Update gate**: Selects information from the past and current input:
$$u_t = \sigma(W_{x_z} x_t + W_{h_z} h_{t-1} + \vec{b}_z) \tag{1.23}$$
2. **Reset gate**: Similarly to update gate, it determines which parts of the previous hidden state are used for further computation:
$$r_t = \sigma(W_{x_r} x_t + W_{h_r} h_{t-1} + \vec{b}_r) \tag{1.24}$$

The **hidden state** is then computed with the use of both gates:

$$h_t = u_t \odot h_{t-1} + (1 - u_t) \odot tanh(W_{x_h} x_t + W_{h_h}(r_t \odot h_{t-1}) + \vec{b}_h) \tag{1.25}$$

As in the previous section, W_x, W_h and $\vec{b}$ with subscripts in the equations above are learned matrices and vectors respectively.

There are also other versions of GRU, where gates are computed using only the previous hidden state h_{t-1} or where update and reset gates are merged together [5].

Same as with standard RNN and LSTM, there is also *bidirectional GRU* (*BGRU*) and is structured in the same way as BLSTM. This architecture was used for task such as sound event detection [25].

1.2.6 Autoencoder

Autoencoder is a type of neural network that can learn a representation of given data by compressing and decompressing the input values. As described in [26], it consists of two parts, the *encoder* and *decoder*. The encoder is typically a dense FNN (other types of neural networks can be used as well) with subsequent layers shrinking in width. The decoder mirrors the structure of encoder with expanding layers. The autoencoder is then trained with a set of data where input matches the target output.

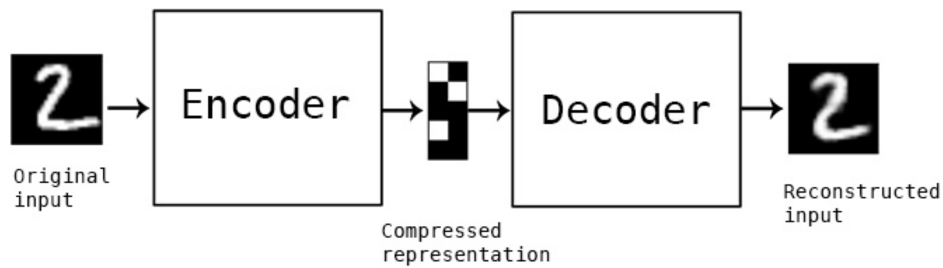


Figure 1.11: Example of autoencoder for digit compression [26]

Although the compressed data could be used in image compression, generally autoencoders don't outperform well-known compression algorithms. Since the compression inside autoencoders isn't lossless (the output is fuzzy), they aren't suited for practical use of image compression.

The places where autoencoders found utilization are dimension reduction and data denoising problems. In dimension reduction, they are used either as a preprocessing stage in machine learning algorithms or before data visualization where large data dimension hinders the comprehension of the image. In data denoising, the autoencoder is trained with noisy images as the input and clear pictures being the output. It isn't limited to images but can also be used with audio and other problems affected by noisiness.

Chapter **2**

Portable Executable

Portable Executable (**PE**) **format** is a file format for Windows operation systems (Windows NT) executables, DLLs (dynamic link libraries) and other programs. Portable denotes the transferability between 32-bit and 64-bit systems. The file format contains all basic information for the OS loader [27]. In this chapter, we will briefly describe the structure of PE file format.

2.1 Structure of PE File

The format of PE file is strictly set as follows, starting with MS-DOS stub and header, followed with file, optional and section headers and finished with program sections as illustrated in Figure 2.1. The detailed description can be found in [28].

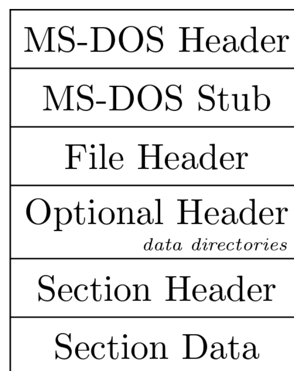


Figure 2.1: Structure of a PE file

2.1.1 MS-DOS Header and Stub

These sections are included for compatibility with past versions of MS-DOS. MS-DOS stub is an executable application which can be run under MS-DOS.

Nowadays, it contains a default program with some variation of the message "This program cannot be run in DOS mode".

2.1.2 File Header

Following is the **signature** which classifies the file as a PE file. Immediately after the signature is located a **file header** containing information such as target machine, the size of the section table, date of file creation, size of optional header and others.

2.1.3 Optional Header

Even though this header is called optional, every executable file has one. This chunk encloses vital info for the loader. Holds information such as the size of code, size of headers, and much more. The last component of this part is a group of **data directories** which consist of information about the import, export, resource, debug along with other tables.

2.1.4 Section Headers

The optional header is directly succeeded with **section headers** (section table). The number of headers (rows of section table) is written in the file header. Each header contains data like virtual address, size and pointer to raw data, and more.

2.1.5 Section Data

Sections are located on addresses specified by their respective headers. The typical sections are:

.text This field incorporates all code segments from the program.

.bss, .rdata, .data Here are stored information about program's variables, constants, ...

.rsrc This section is structured into a resource tree which holds resource info for an application.

.edata Data which application exports, usually found in DLLs.

.idata Opposite to .edata, this section contains imported data.

.pdata Here we can find function entries for exception handling.

.debug Any debug information from the compiler.

Chapter **3**

Experiments

As mentioned in chapter Introduction, the goal of this thesis is to found the most appropriate structure of LSTM network for static malware detection. For this task, we used Python as programming language together with interactive computational environment Jupyter Notebook [29]. We had chosen this language because it's well-known inside the machine learning community and for its ease of use and comprehension. We used several packages from Python `SciPy` ecosystem [30], such as `NumPy`, `pandas` or `Matplotlib`.

The implementation process consists of multiple stages, which were briefly mentioned in the Introduction. In the following sections, we thoroughly describe each part of our coding process, starting with PE file collection, through feature engineering and finishing with the central part, testing different RNN architectures.

3.1 Dataset Description

For training and testing, we have collected two datasets. A smaller dataset which we gathered from available resources and EMBER dataset [31] from Github.

3.1.1 Our Dataset

A smaller dataset consists of 30,154 samples which are evenly distributed between malware and benign files. For amassing benign files, we searched disks on university computers and the malware files were obtained from online repository `https://virusshare.com` which we thanks for access.

3.1.2 EMBER Malware Dataset

The **EMBER** (Endgame Malware BEnchmark for Research) [31] dataset is a benchmark dataset for malware detection researches. We used the 2018 version

which contains one million samples which are divided into train and test sets. The train set consists of 800,000 samples with 200,000 samples without label and the rest equally split between malware and benign programs. We left out the unlabeled samples and worked only with the remaining 600,000 labeled data. The test set contains 200,000 samples which are evenly partitioned amid malware and benign. Each row of the dataset contains features extracted from PE files with over 2,300 columns. It contains attributes such as hashed printable strings, header and sections information or entries from import and export tables.

3.2 Feature Extraction

For extracting features from PE files, we used Python module `pefile` [32]. This module extracts all PE file attributes into an object from which they can be easily accessed. The structure of the PE file is explained in the chapter 2. We tried to use as much data as possible and reached the total number of features of 303. Features can be divided into multiple categories based on its origin from the PE file.

Headers Data from DOS, NT, File and Optional headers.

Data Directories Names and sizes of all data directories. Also adding detailed information from prevalent directories for instance IMPORT, EXPORT, RESOURCE and DEBUG directories.

Sections Names, sizes, entropies of all PE sections expressed by their average, min, max, mean and standard deviation. To cooperate with a variable amount of section in different files, we decided to describe only the first four and last sections individually.

Others Extra characteristics associated with a file, e.g. byte histogram, printable strings or version information.

To see the code for this part look at `features_extractor.ipynb` on the enclosed DVD.

3.3 Feature Preparation

Since machine learning models typically work with numerical features, we must encode strings and other categorical data into numeric values. This strategy is necessary for more than 60 out of 303 columns. We chose to perform common transformation techniques on the entire dataset as opposed to only using the training set. We believe that by doing so, we can better focus on designing RNN architectures and our results won't be affected by the capability of other algorithms.

3.3.1 Vectorization

Upfront, we transformed string features into sparse matrix representation using `TfidfVectorizer` from the `scikit-learn` Python library [33]. This class demands *corpus* (collection of documents) as an input. We also adjusted parameters `stop_words` and `max_df` that influence which words to exclude from further calculations, because they are either commonly used in a given language and don't bear any meaning, or occur with such high frequencies that they aren't statistically interesting for us. To eliminate the massive rise of dimensionality, we set `max_features` parameter according to the feature's cardinality. The transformation itself consists of converting sentences to vectors of token counts. Then they are transformed into tf-idf representation. **Tf-idf** is an abbreviation for *term frequency times inverse document frequency*. It is a way to express the weight of a single word in the corpus [34].

Term frequency is the frequency of a word inside the document. The formula is:

$$tf(w,d) = \frac{n_{w,d}}{\sum_k n_{k,d}} \tag{3.1}$$

Where $n_{w,d}$ is the number of times word w appears in document d and the denominator is the sum of all words found in d.

Inverse document frequency is a scale of how much a word is rare across the whole corpus:

$$idf(w,D) = \log \frac{|D|}{|d \in D : w \in d|} \tag{3.2}$$

It is a fraction of the total number of documents in corpus D divided by the number of documents containing the specific word.

Tf-idf is then calculated as a multiplication of these two values [35]:

$$\text{tf-idf}(w,d) = tf(w,d) \cdot idf(w,D) \tag{3.3}$$

All of this is done by the aforementioned class `TfidfVectorizer`, and as a result, we get a matrix of tf-idf features which can be used in further computations.

3.3.2 Hashing

For non-string values, we used a technique called **feature hashing**. This approach turns the column of values into a sparse matrix using the value's hash as an index to the matrix. For this task, we used `FeatureHasher` also from `scikit-learn`. The class takes as an optional argument `n_features` which limits the number of columns in the output matrix. We set this argument dynamically according to the size of the feature's value set.

Full code with vectorization and feature hashing is available in notebook `features_transformation.ipynb` on the attached DVD.

3.4 Feature Selection

Even though we tried to limit the rise of new features, we ended up with 1488 columns. To speed up the forthcoming training process, we tried several feature selection techniques to reduce the dimensionality of the dataset.

Before all else, we filled missing values by column's mean and divided data into train and test splits to ensure correct evaluation of the model's performance. For this, we used `train_test_split` from `sklearn.model_selection` with test split taking 20% of the dataset. Afterwards, we transformed features to stretch across a smaller range. For this task, we looked for another class from `sklearn.preprocessing` library and selected `MinMaxScaler`. This scaler turns each feature x to lie between zero and one. The transformation is calculated as:

$$\frac{x_i - \min(x)}{\max(x) - \min(x)} \tag{3.4}$$

For feature selection, we tried multiple methods. Namely, PCA (Principal Component Analysis), autoencoder and `SelectFromModel` . All selectors were limited to the maximum number of 200 features. The first two mentioned methods were outperformed by `SelectFromModel` from `scikit-learn`, thus we used it in future tests. Full results are shown in Table 3.1 below.

Table 3.1: Comparison of different selection techniques

technique	ACC	TPR	FPR
Autoencoder	0.8420	0.8169	0.1332
PCA	0.9783	0.9727	0.0162
SelectFromModel (ExtraTree)	0.9803	0.9730	0.0125
SelectFromModel (RandomForest)	0.9823	0.9780	0.0135
SelectFromModel (LogisticRegression)	0.9823	0.9780	0.0135

`SelectFromModel` is a selector that picks features based on their importance. The importance is established by an *estimator* which can be, for example, *extra-trees* classifier or *logistic regression*. We chose *random forest* classifier as it had high *accuracy* while relatively low *false-positive rate*. More about evaluation metrics can be found in 4.1. This classifier works as an ensemble of decision trees. The trees are fitted on a random subgroup of samples, and results are then averaged to get the final answer.

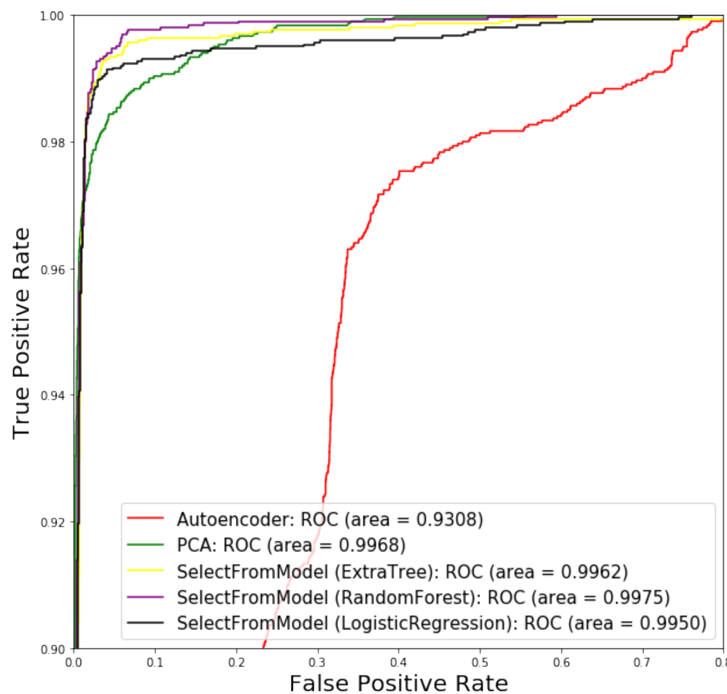


Figure 3.1: ROC curve of different feature selection techniques

Table 3.1 and Figure 3.1 describe the performance of LSTM network on datasets created by different selection techniques.

The source code can be found in `features_selection.ipynb` located on enclosed DVD.

3.5 Proposed Method

For modelling various types of recurrent neural networks, we chose `Keras`, the Python deep learning library [36]. To explore the performance of different RNN architectures, we ran widespread testing from which we selected the best performing designs. With those, we ran further experiments with slight adjustments to fine-tune their structure to maximize the classification performance.

3.5.1 Approximate Testing

The experimentation consisted of testing more than 300 different architectures of RNN on our training dataset, which we further split into train and validation groups. Training each model up to 50 epochs with the batch size of 128. Entire testing was run with models compiled under Adam optimizer. *Adam*, an abbreviation of adaptive moment estimation, is an optimization algorithm

which computes adaptive learning rates for each parameter. It was presented by Kingma and Ba in 2014 [37].

We analysed standard RNN (1.2.3), LSTM (1.2.4) and GRU (1.2.5) units and also included bidirectional version of these architectures. We tried various heights and widths of the networks. With the number of units in layer ranging between 1 and 128 and the number of stacked layers from 1 up to 7. We also experimented with added dropout regularization between layers. *Dropout* is a technique used during the training process of a neural network. It makes changes to random sub-sample of neural network which reduce net's overfitting to the training data [11].

From these trials, we selected structures which performed the best on the validation set. The results for the top performing nets can be found in Table 3.2. The results are sorted by accuracy (ACC), the last column shows training time (in seconds) needed to achieve these results.

Table 3.2: The most promising RNN architectures

type	bidirectional	num_layers	num_units_in_layer	ACC	TPR	FPR	train_time (s)
LSTM	True	5	16	0.9853	0.9835	0.0129	141.0
LSTM	True	6	32	0.9851	0.9839	0.0137	157.9
GRU	True	5	32	0.9849	0.9859	0.0162	155.3
LSTM	False	5	64	0.9847	0.9789	0.0096	84.2
GRU	False	3	32	0.9847	0.9847	0.0154	64.4
LSTM	True	7	32	0.9847	0.9797	0.0104	185.7
GRU	True	4	32	0.9845	0.9855	0.0166	130.6
LSTM	False	4	32	0.9845	0.9810	0.0121	73.6
LSTM	True	3	64	0.9845	0.9814	0.0125	93.0
GRU	True	6	16	0.9845	0.9818	0.0129	174.8
LSTM	True	5	32	0.9842	0.9868	0.0183	139.5
LSTM	True	2	128	0.9840	0.9843	0.0162	71.6
GRU	False	6	64	0.9840	0.9835	0.0154	107.2
GRU	True	4	16	0.9840	0.9797	0.0116	125.8
GRU	True	6	64	0.9840	0.9826	0.0145	177.1

3.5.2 Promising Structures

The structures with the most potential we farther tuned-up in order to boost their performance. We found that LSTM and GRU networks performed well, while standard RNN lagged behind with only one representative found at the end of top 50. Henceforward, we focused mainly on LSTM networks as they had the best results overall. The ubiquitous trend between tested networks has shown that deeper architectures seem to comprehend data better, yet there were huge fluctuation in achieved FPR.

From Table 3.2, we selected structures with the best accuracy which was more than 98% for the most performing networks. We reran the first five architectures through our benchmark, but now up to 200 epochs to see if they could benefit from further learning.

Table 3.3: Top five RNN architectures after 200 epochs

type	bidirectional	num_layers	num_units_in_layer	ACC	TPR	FPR	train_time (s)
LSTM	True	5	16	0.9834	0.9773	0.0104	555.5
LSTM	True	6	32	0.9845	0.9785	0.0096	624.3
GRU	True	5	32	0.9855	0.9806	0.0096	593.2
LSTM	False	5	64	0.9859	0.9814	0.0096	332.7
GRU	False	3	32	0.9842	0.9785	0.0100	251.0

From the results in Table 3.3, we can see that some architectures increased their accuracy while some didn't improve further. Where as some networks recorded rising TPR, it was usually at the expense of higher FPR and vice versa.

We carried more experiments, training networks up to 2,000 epochs, adding extra hidden layers or using different optimizers. In the end, we came up with a combination of *dense* (layer of FNN) and LSTM layers, compiled with Adam optimizer that achieved reasonably good results on our dataset.

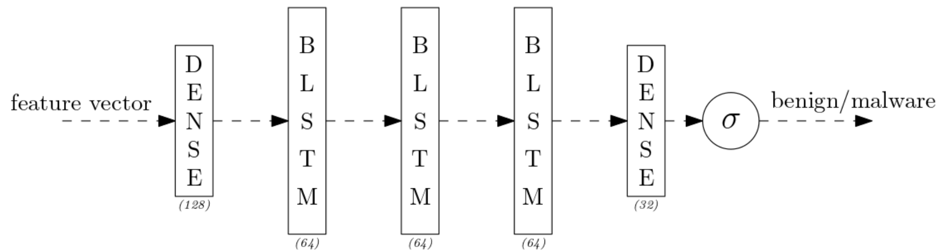


Figure 3.2: Proposed structure of LSTM network

In Figure 3.2, we loosely illustrated the architecture of the best performing LSTM network we found. Combining dense neural layers with bidirectional LSTM. The output layer consists of sigmoid neuron unit, as it's typical for binary classification problems. In addition to the image, the layers are interleaved with the dropout set to 35%.

The graph in Figure 3.3 illustrates how our LSTM architecture comprehended the dataset over 700 epochs. We can see that accuracy remained similar during the whole training process. Model maintained low FPR until 200 epochs after which started to overfit to positive samples and increased TPR at the cost of higher FPR. For our task the malware detection, the sweet spots are located where we can found nearly the highest accuracy while maintaining a low false-positive rate.

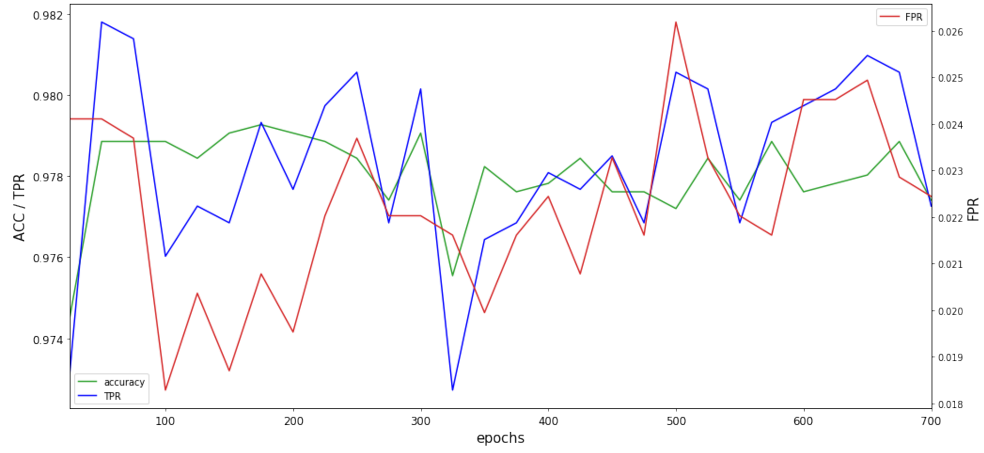


Figure 3.3: Evolution of ACC, TPR and FPR over 700 epochs on the validation set

TN 49.78	*FP* 0.50
FN 1.09	*TP* 48.63

Figure 3.4: Confusion matrix presenting our results

The results on the test set after 250 epochs are shown by the confusion matrix in Figure 3.4.

Chapter **4**

Results

In this chapter, we analysed the performance of our LSTM architecture from Chapter 3 and compared it with other machine learning models.

In the first place, we describe the metrics used for model evaluation. Next, we evaluate our LSTM architecture against other supervised machine learning algorithms. In the end, we compare our results with related work on the topic of static malware detection using neural networks.

4.1 Evaluation Metrics

The crucial part of designing machine learning algorithms is evaluation. Evaluation needs to be standardized yet personalized to the problem. In our case, malware detection, the target is to detect as much malware as possible, while minimizing the number of false alarms (false positives).

4.1.1 Confusion Matrix

The basis for evaluating classifiers is the **confusion matrix**. This matrix is used as a foundation for calculating other metrics such as *accuracy*, *precision*, *sensitivity*, *specificity* or *ROC Curve* [38].

True Positive Prediction says positive, and it's correct.

True Negative Prediction says negative, and it's correct.

False Positive Prediction says positive but actual value is negative.

False Negative Prediction says negative but actual value is positive.

	0	1
0	TN	FP *Type 1 Error*
1	FN *Type 2 Error*	TP

Actual Values (rows) / Predicted Values (columns)

Figure 4.1: Confusion matrix, 0 stands for negative and 1 for positive values

Figure 4.1 shows the composition of a confusion matrix where each field contains the number of samples belonging to a given class. The matrix can be extended to cover more than 2 classes for multi-classification problems. For our case, we stay with binary classification.

Many metrics can be calculated from the confusion matrix, for instance:

Accuracy (ACC) Proportion of correctly classified samples out of all predictions:

$$ACC = \frac{TP + TN}{TP + TN + FP + FN} \tag{4.1}$$

Precision (PPV) Ratio of true positives to all samples predicted as positive:

$$PPV = \frac{TP}{TP + FP} \tag{4.2}$$

Sensitivity (*TPR, Recall*) How many samples from the positive class were predicted correctly:

$$TPR = \frac{TP}{TP + FN} \tag{4.3}$$

Combination of precision and recall is called **f-score**, which maximizes both metrics.

Specificity Alike sensitivity, number of samples from the negative class which were classified correctly:

$$\text{Specificity} = \frac{TN}{TN + FP} \tag{4.4}$$

Fall-out (*FPR*) Probability of predicting samples from the negative class as positives:

$$FPR = \frac{FP}{FP + TN} = 1 - \text{Specificity} \tag{4.5}$$

In the definitions above, the TN, FP, FN and TP refers to the number of samples belonging to their respective class [39].

4.1.2 ROC Curve

ROC curve, shorten from receiver operating characteristic curve is a graph comparing the FPR on the x-axis with TPR on the y-axis at different classification thresholds. An example of ROC curve can be seen in Figure 3.1.

To evaluate and compare the performance of the model, we can calculate the area under the curve (**AUC**). The area ranges between zero and one. A model with AUC equal to zero means that all predictions were wrong. In fact, it means that model is predicting exactly opposite values. Area of the size one denotes perfect model, and random predictions would lead to an area of 0.5 [40]. It is important to understand that AUC score is only useful in problems where we don't prioritize prediction of either class. In the field of malware detection, the goal is typically to maximize accuracy while minimizing false-positive rate.

4.2 Supervised ML Algorithms

In this section, we present the performance of our architecture against other supervised machine learning algorithms. After multiple testing stages, we found the *random forest* classifier to consistently outperform simple *decision trees*, *k-nearest neighbours*, *logistic regression* and other classifiers from the `scikit-learn` library we tried. Possible explanation of superb *random forest* and other ensemble models results might lie in the feature selection approach we used. The `SelectFromModel` technique selected features based on importance weights computed by the given estimator, as which we used the `RandomForestClassifier`.

Table 4.1: Comparison of our LSTM network with well-known supervised machine-learning algorithms

ML algorithm	accuracy	TPR	FPR	TN	FP	FN	TP
LSTM	0.9841	0.9780	**0.0099**	**0.4978**	**0.0050**	0.0109	0.4863
Feed-ForwardNetwork	0.9859	0.9840	0.0122	0.4966	0.0061	0.0080	0.4893
DecisionTree	0.9851	0.9870	0.0168	0.4943	0.0085	0.0065	0.4908
RandomForest	**0.9884**	**0.9897**	0.0129	0.4963	0.0065	**0.0051**	**0.4921**
AdaBoost	0.9801	0.9807	0.0204	0.4925	0.0103	0.0096	0.4876
kNN	0.9778	0.9763	0.0208	0.4923	0.0104	0.0118	0.4855
NaiveBayesGaussian	0.9275	0.8756	0.0211	0.4921	0.0106	0.0618	0.4354
LogisticRegression	0.9650	0.9617	0.0317	0.4868	0.0159	0.0191	0.4782

From the Table 4.1 we can see that, even though our LSTM architecture didn't score the best in terms of accuracy, it achieved the lowest FPR with only around 0.5% samples from entire dataset classified as false positive. Figure 4.2 below compares ROC curves of these algorithms.

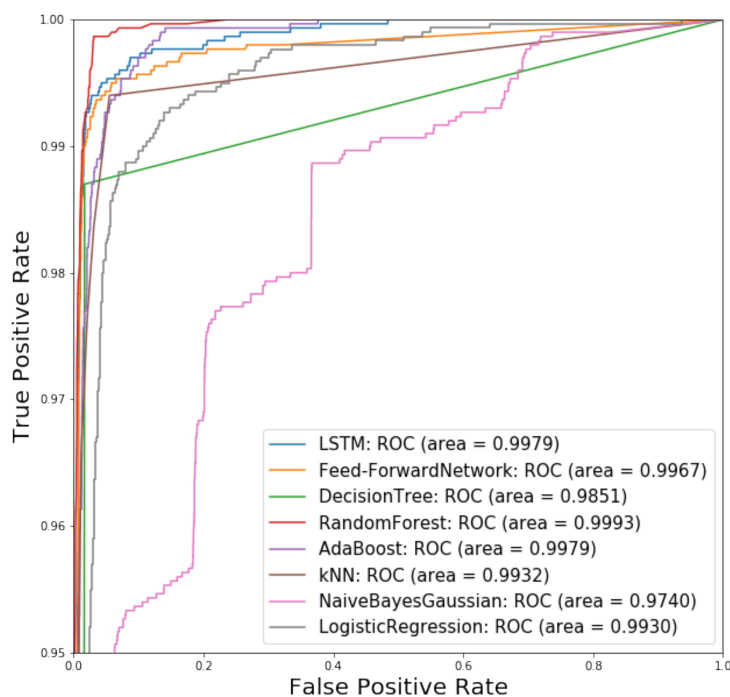


Figure 4.2: ROC curve of tested ML algorithms

The results above were carried out on our dataset described in 3.1.1. Later, we reran all models on EMBER dataset described in 3.1.2. We didn't optimize our approach in the same way as we did for our dataset, so we used the same standardization and feature selection techniques as described in chapter 3.

Table 4.2: Comparison of our LSTM network with well-known supervised machine-learning algorithms (EMBER)

ML algorithm	accuracy	TPR	FPR	TN	FP	FN	TP
LSTM	0.9452	0.9387	0.0483	0.4758	0.0242	0.0307	0.4693
Feed-ForwardNetwok	0.9440	0.9463	0.0583	0.4709	0.0291	0.0268	0.4732
DecisionTree	0.9077	0.9066	0.0912	0.4544	0.0456	0.0467	0.4533
RandomForest	**0.9525**	**0.9502**	**0.0452**	**0.4774**	**0.0226**	**0.0249**	**0.4751**
AdaBoost	0.8327	0.9045	0.2391	0.3805	0.1195	0.0478	0.4522
kNN	0.9204	0.9103	0.0695	0.4653	0.0347	0.0448	0.4552
NaiveBayesGaussian	0.6707	0.5919	0.2506	0.3747	0.1253	0.2040	0.2960
LogisticRegression	0.7817	0.8952	0.3317	0.3341	0.1659	0.0524	0.4476

The results from Table 4.2 didn't end up as great as results with our dataset shown in Table 4.1. But as we mentioned before, we didn't optimize our approach to this dataset. In Figure 4.3 below you can see a comparison of ROC curves on EMBER dataset.

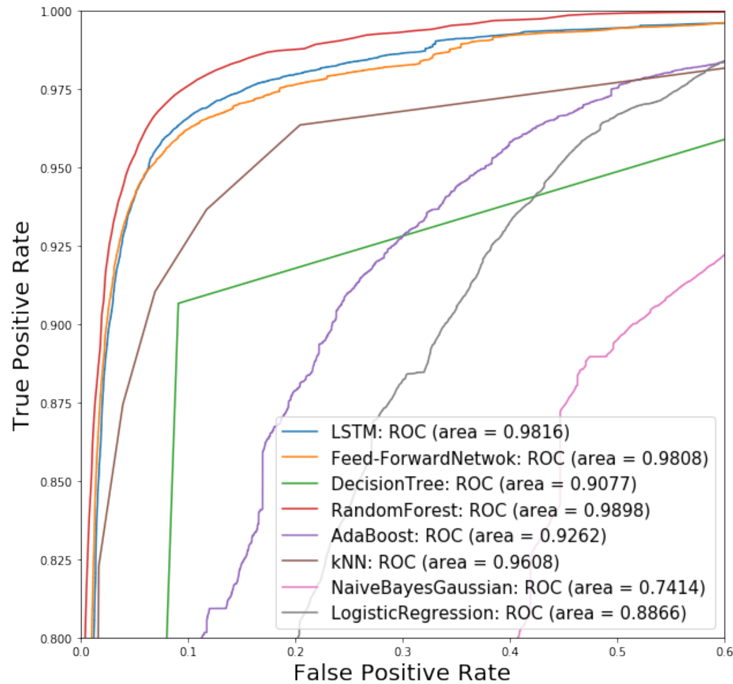


Figure 4.3: ROC curve of tested ML algorithms (EMBER)

4.3 Comparison with the State-of-the-art

In this part, we compare our results with related research in the field of static malware detection. We focused on the papers linked to neural networks, notably recurrent neural networks.

In [41], they used opcodes (operation code, part of machine language instruction [42]) extracted from a disassembled binary file. From those opcodes, they created a language with the help of word embedding. *Word embedding* is a technique consisting of converting words to vector representations. This method groups similar words together, which helps in training of natural language processing models [43]. The language is then processed by LSTM network to get the prediction. They achieved an AUC-ROC score of 0.99, but their dataset consisted of only 1,092 samples.

Much larger dataset of 90,000 samples was used in [44]. They used a LSTM network to process API call sequences combined with CNN to detect malicious files. While also using dynamic and static features, they managed an accuracy of 97.3%.

Deep neural networks were also used in [45] with the help of Bayesian statistics. They worked with a large dataset of more than 400 thousands binaries. With fixed FPR at 0.1%, they reported AUC-ROC of 0.99964 with TPR of 95.2%.

Hardy et al. in [46] used stacked autoencoders for malware classification and achieved an accuracy of 95.64% on 50,000 samples.

In terms of attitude to malware detection, Vinayakumar et al. [47] are the closest to our solution. They trained stacked LSTM network and achieved an accuracy of 97.5% with AUC-ROC score of 0.998. That said they focused on android file and collected only 558 APKs (Android application package).

It's hard to compare our results with these works due to everyone using different datasets for evaluation. We tried to confront this problem by also evaluating our proposed method on freely available, EMBER dataset. While using our dataset, we can see that in terms of accuracy, our combination of dense and stacked LSTM layers surpassed most of the other works, while maintaining a low number of falsely classified benign files. However, on EMBER dataset, our results didn't come out as brilliant, but as we mentioned earlier, we didn't optimize our approach to this dataset and more research needs to be done in this field.

Conclusion

The goal of this thesis was to study various RNN architectures with a focus on LSTM and apply them in static malware detection. We collected a large number of PE binaries from available resources. From these binaries, we extracted as many features as possible. With these features, we ran feature selection algorithms to reduce the dimension of our dataset. Later, we conducted extensive testing with the goal of founding the most appropriate RNN architectures, which we further fine-tuned to enhance their performance. In the end, we compared our best performing architecture with other supervised ML models and related research.

We found that bidirectional LSTM neural networks outperformed other tested recurrent neural networks, notably standard and gated recurrent units. We experimented with various structures of LSTM networks and came up with a combination of dense and LSTM layers. The input firstly goes through the fully connected dense layer, which is followed by stacked bidirectional LSTM layers with 64 units each and finished with another dense layer and output neuron with the sigmoid activation function.

This structure achieved an accuracy of 98.41%, with 97.80% TPR and 0.99% FPR after being trained on our training dataset for 250 epochs.

To summarize this work, we managed to train the LSTM network to achieve reasonably good detection rate at only 0.5% false positives. Thus we proved that RNNs could be used as a reliable malware detection method. However, more improvement can be made, especially in the feature engineering of PE files, where we foresee vast improvements. Also, we limited our research to only static analysis and didn't study LSTM performance while using dynamic features such as API calls, which could lead to further improvements.

Bibliography

1. AV-TEST. *SECURITY REPORT 2018/19* [online]. 2019 [visited on 04/30/2020]. Available from: https://www.av-test.org/fileadmin/pdf/security_report/AV-TEST_Security_Report_2018-2019.pdf.
2. SYMANTEC, C. *Internet security threat report 2019* [online]. 2014 [visited on 04/30/2020]. Available from: https://www-west.symantec.com/content/dam/symantec/docs/reports/istr-24-2019-en.pdf.
3. LATTO, Nica. *What is WannaCry?* [online]. 2020 [visited on 04/30/2020]. Available from: https://www.avast.com/c-wannacry.
4. CLOONAN, John. Advanced malware detection-signatures vs. behavior analysis. *Infosecurity Magazine*. 2017, vol. 11. ISSN 1878-741X.
5. BENGIO, Yoshua; GOODFELLOW, Ian; COURVILLE, Aaron. *Deep learning*. Citeseer, 2017. ISBN 0262035618.
6. MCCULLOCH, Warren S; PITTS, Walter. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*. 1943, vol. 5, no. 4, pp. 115–133. ISSN 0007-4985.
7. KRISHTOPA. *What Are Neural Networks, Why They Are So Popular And What Problems Can Solve* [online]. 2016 [visited on 03/23/2020]. Available from: https://steemit.com/academia/@krishtopa/what-are-neural-networks-why-they-are-so-popular-and-what-problems-can-solve.
8. ROJAS, Raúl. *Neural networks: a systematic introduction*. Springer Science & Business Media, 2013. ISBN 9783642610684.
9. NIELSEN, Michael A. *Neural networks and deep learning* [online]. Determination press San Francisco, CA, USA: 2015 [visited on 12/26/2019]. Available from: http://neuralnetworksanddeeplearning.com/.

10. OTFRIED, Cheong. *Ipe Drawing Editor* [online]. 2019. 7.2.13/7 [visited on 12/26/2019]. Available from: `http://ipe.otfried.org/`.

11. LESKOVEC, Jure; RAJARAMAN, Anand; ULLMAN, Jeffrey David. *Mining of massive data sets*. Cambridge university press, 2020. ISBN 9781108476348.

12. LIPTON, Zachary C; BERKOWITZ, John; ELKAN, Charles. A critical review of recurrent neural networks for sequence learning. *arXiv preprint arXiv:1506.00019* [online]. 2015, pp. 5–25 [visited on 12/20/2019]. ISSN 2331-8422. Available from: `https://arxiv.org/pdf/1506.00019.pdf`.

13. BLUM, Avrim L; RIVEST, Ronald L. Training a 3-node neural network is NP-complete. In: *Machine learning: From theory to applications*. Springer, 1993, pp. 9–28. ISBN 9783540564836.

14. GRAVES, Alex. Supervised sequence labelling. In: *Supervised sequence labelling with recurrent neural networks*. Springer, 2012, pp. 13–39. ISBN 9783642247965.

15. SCHUSTER, Mike; PALIWAL, Kuldip K. Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing*. 1997, vol. 45, no. 11, pp. 2673–2681. ISSN 1053-587X.

16. HOCHREITER, Sepp; SCHMIDHUBER, Jürgen. Long short-term memory. *Neural computation*. 1997, vol. 9, no. 8, pp. 1735–1780. ISSN 0899-7667.

17. ZAREMBA, Wojciech; SUTSKEVER, Ilya; VINYALS, Oriol. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329* [online]. 2014, pp. 1–3 [visited on 12/15/2019]. ISSN 2331-8422. Available from: `https://arxiv.org/pdf/1409.2329.pdf`.

18. GRAVES, Alex; SCHMIDHUBER, Jürgen. Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural networks*. 2005, vol. 18, no. 5-6, pp. 602–610. ISSN 2161-4393.

19. BROWNLEE, Jason. How to Develop a Bidirectional LSTM For Sequence Classification in Python with Keras [online]. 2019 [visited on 05/30/2020]. Available from: `https://machinelearningmastery.com/develop-bidirectional-lstm-sequence-classification-python-keras/`.

20. LIWICKI, Marcus; GRAVES, Alex; FERNÀNDEZ, Santiago; BUNKE, Horst; SCHMIDHUBER, Jürgen. A novel approach to on-line handwriting recognition based on bidirectional long short-term memory networks. In: *Proceedings of the 9th International Conference on Document Analysis and Recognition, ICDAR 2007* [online]. 2007 [visited

on 05/29/2020]. Available from: http://www.cs.toronto.edu/~graves/icdar_2007.pdf.

21. GRAVES, Alex; LIWICKI, Marcus; FERNÁNDEZ, Santiago; BERTOLAMI, Roman; BUNKE, Horst; SCHMIDHUBER, Jürgen. A novel connectionist system for unconstrained handwriting recognition. *IEEE transactions on pattern analysis and machine intelligence*. 2008, vol. 31, no. 5, pp. 855–868. ISSN 0162-8828.
22. KIPERWASSER, Eliyahu; GOLDBERG, Yoav. Simple and Accurate Dependency Parsing Using Bidirectional LSTM Feature Representations. *Transactions of the Association for Computational Linguistics* [online]. 2016, vol. 4, pp. 313–327 [visited on 05/30/2020]. ISSN 2307-387X. Available from DOI: 10.1162/tacl_a_00101.
23. SUNDERMEYER, Martin; ALKHOULI, Tamer; WUEBKER, Joern; NEY, Hermann. Translation Modeling with Bidirectional Recurrent Neural Networks. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)* [online]. Doha, Qatar: Association for Computational Linguistics, 2014, pp. 14–25 [visited on 05/30/2020]. Available from DOI: 10.3115/v1/D14-1003.
24. JIANG, Junshu; ZOU, Shangjie; SUN, Yu; ZHANG, Shengxiang. GL-BLSTM: a novel structure of bidirectional long-short term memory for disulfide bonding state prediction. *arXiv preprint arXiv:1808.03745* [online]. 2018 [visited on 05/29/2020]. ISSN 2331-8422. Available from: https://arxiv.org/pdf/1808.03745.pdf.
25. LU, Rui; DUAN, Zhiyao. Bidirectional GRU for sound event detection. *Detection and Classification of Acoustic Scenes and Events*. 2017. ISSN 2329-9290.
26. CHOLLET, Francois. *Building autoencoders in keras* [online]. 2016 [visited on 04/16/2020]. Available from: https://blog.keras.io/building-autoencoders-in-keras.html.
27. KOWALCZYK, Krzysztof. *Portable Executable File Format* [online]. 2018 [visited on 04/05/2020]. Available from: https://blog.kowalczyk.info/articles/pefileformat.html.
28. KARL BRIDGE, Microsoft. *PE Format - Win32 apps* [online]. 2019 [visited on 04/05/2020]. Available from: https://docs.microsoft.com/en-us/windows/win32/debug/pe-format.
29. KLUYVER, Thomas et al. Jupyter Notebooks-a publishing format for reproducible computational workflows. In: *ELPUB* [online]. 2016, pp. 87–90 [visited on 01/05/2020]. Available from DOI: http://eprints.soton.ac.uk/id/eprint/403913.

30. VIRTANEN, Pauli et al. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* [online]. 2020, vol. 17, pp. 261–272 [visited on 03/30/2020]. ISSN 1548-7105. Available from DOI: https://doi.org/10.1038/s41592-019-0686-2.

31. ANDERSON, H. S.; ROTH, P. EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models. *ArXiv e-prints* [online]. 2018 [visited on 02/25/2020]. ISSN 2331-8422. Available from arXiv: 1804.04637 [cs.CR].

32. CARRERA, E. *Pefile* [online]. 2017 [visited on 01/15/2020]. Available from: https://github.com/erocarrera/pefile.

33. PEDREGOSA, F. et al. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*. 2011, vol. 12, pp. 2825–2830. ISSN 1533-7928.

34. MAKLIN, Cory. *TF IDF: TFIDF Python Example* [online]. 2019 [visited on 04/25/2020]. Available from: https://towardsdatascience.com/natural-language-processing-feature-engineering-using-tf-idf-e8b9d00e7e76.

35. KLOUDA, Karel; VAŠATA, Daniel. *Neuronové sítě: NLP* [online]. 2019 [visited on 04/25/2020]. Available from: https://courses.fit.cvut.cz/BI-VZD/lectures/files/BI-VZD-11-cs-handout.pdf.

36. CHOLLET, François et al. *Keras* [https://keras.io]. 2015 [visited on 03/02/2020].

37. KINGMA, Diederik P; BA, Jimmy. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* [online]. 2014 [visited on 04/27/2020]. ISSN 2331-8422. Available from: https://arxiv.org/pdf/1412.6980.pdf.

38. NARKHEDE, Sarang. *Understanding Confusion Matrix* [online]. 2019 [visited on 05/29/2020]. Available from: https://towardsdatascience.com/understanding-confusion-matrix-a9ad42dcfd62.

39. MARKHAM, Kevin. *Simple guide to confusion matrix terminology* [online]. 2014 [visited on 04/16/2020]. Available from: https://www.dataschool.io/simple-guide-to-confusion-matrix-terminology/.

40. BROWNLEE, J. *How and when to use ROC curves and precision-recall curves for classification in Python* [online]. 2018 [visited on 04/17/2020]. Available from: https://machinelearningmastery.com/roc-curves-and-precision-recall-curves-for-classification-in-python/.

41. LU, Renjie. Malware Detection with LSTM using Opcode Language. *arXiv preprint arXiv:1906.04593* [online]. 2019 [visited on 11/15/2019]. ISSN 2331-8422. Available from: https://arxiv.org/pdf/1906.04593.pdf.

42. BARRON, David William. *Assemblers and loaders*. Elsevier Science Inc., 1978. ISBN 9780130525642.

43. MIKOLOV, Tomas; SUTSKEVER, Ilya; CHEN, Kai; CORRADO, Greg S; DEAN, Jeff. Distributed representations of words and phrases and their compositionality. In: *Advances in neural information processing systems*. 2013, pp. 3111–3119. ISBN 9781632660244.

44. ZHOU, Huan. Malware Detection with Neural Network Using Combined Features. In: *China Cyber Security Annual Conference* [online]. 2018, pp. 96–106 [visited on 11/15/2019]. Available from: https://link.springer.com/chapter/10.1007/978-981-13-6621-5_8.

45. SAXE, Joshua; BERLIN, Konstantin. Deep neural network based malware detection using two dimensional binary program features. In: *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*. 2015, pp. 11–20. ISBN 9781509003174.

46. HARDY, William; CHEN, Lingwei; HOU, Shifu; YE, Yanfang; LI, Xin. DL4MD: A deep learning framework for intelligent malware detection. In: *Proceedings of the International Conference on Data Mining (DMIN)*. 2016, p. 61. ISBN 9781601324313.

47. VINAYAKUMAR, R; SOMAN, KP; POORNACHANDRAN, Prabaharan; SACHIN KUMAR, S. Detecting Android malware using long short-term memory (LSTM). *Journal of Intelligent & Fuzzy Systems*. 2018, vol. 34, no. 3, pp. 1277–1288. ISSN 1875-8967.

Appendix A

Acronyms

ACC Accuracy

AUC Area Under Curve

BGRU Bidirectional Gated Recurrent Unit

BLSTM Bidirectional Long Short-Term Memory

BPTT Backpropagation Through Time

BRNN Bidirectional Recurrent Neural Network

CNN Convolution Neural Network

FFN Feed-Forward Network

FN False Negative

FP False Positive

FPR False Positive Rate

GRU Gated Recurrent Unit

LSTM Long Short-Term Memory

ML Machine Learning

NN Neural Network

PCA Principal Component Analysis

PE Portable Executable

PPV Positive Predictive Values

ReLU Rectified Linear Unit

RNN Recurrent Neural Network

ROC Receiver Operating Characteristic

tanh Hyperbolic Tangent

TN True Negative

TP True Positive

TPR True Positive Rate

Appendix B

Contents of enclosed CD

- README.md .... the Markdown file with description
- src .... the directory of implementation source codes
  - models .... the directory of trained model
  - resources .... the directory of datasets
  - results .... the directory of results
- text .... the directory of thesis text and LaTeX source codes
  - bib .... the directory of bibliography files
  - fig .... the directory of figures
  - tex .... the directory of LaTeX source codes
  - presentation .... the directory of thesis presentation
  - BP_Kozak_Matous_2020.pdf .... the thesis in PDF format