Assignment of master’s thesis

Title: Malware Detection Using Visualization Techniques
Student: Bc. Ihor Salov
Supervisor: Mgr. Olha Jurečková
Study program: Informatics
Branch / specialization: Computer Security
Department: Department of Information Security
Validity: until the end of summer semester 2023/2024

Instructions

Today, there are many different methods for analyzing and detecting malware. Deep learning and image processing have achieved a good effect on malware classification. A technique introduced in 2008 suggested that malware binaries can be converted into grayscale or RGB (Red, Green, Blue) color images. This representation can be used to identify each type of malware specifically. Different types of malware of the same family have been observed to have identical image structures when converted from binary. Models using deep learning convolutional neural networks (CNN) can embrace images as input simply.

Instructions:
1. Study the existing approaches of malware detection procedures/techniques and review previous research on malware detection using visualization.
2. Implement or use an existing library of current analysis techniques of such visualizations that utilize machine learning and neural networks.
3. Choose an appropriate publicly available dataset and train a neural network able to process the created images and classify them as malicious or benign.
4. Discuss the results and compare them with other malware detection techniques.

Electronically approved by prof. Ing. Róbert Lorencz, CSc. on 26 January 2023 in Prague.
MALWARE DETECTION USING VISUALIZATION TECHNIQUES

Bc. Ihor Salov
## Contents

### Declaration  vii

### Abstract viii

### Abbreviations ix

### Introduction 1

1. Research 3
   1.1 Related work 3

2. Malware visualization 13
   2.1 Image format 13
   2.2 Malware to image conversion 14

3. Convolutional Neural Networks 17
   3.1 Building blocks 17
      3.1.1 Convolution 18
      3.1.2 Padding 19
      3.1.3 Strides 19
      3.1.4 Pooling 20
      3.1.5 Activation function 20
      3.1.6 Fully connected layer 21
   3.2 Training 22
      3.2.1 Loss function 22
      3.2.2 Gradient descent 23
      3.2.3 Dropout 25
      3.2.4 Cross-validation 25
      3.2.5 Backpropagation in CNN 25

4. Datasets 29
   4.1 SOREL-20M 29
      4.1.1 GitHub repository 29
      4.1.2 AWS S3 bucket 29
      4.1.3 Metadata structure 30
   4.2 Benign samples 31
   4.3 SORRY dataset 31
   4.4 Malevis 33

5. Implementation 37
   5.1 Prerequisites 37
      5.1.1 Transfer learning 37
      5.1.2 Classifier selection 37
   5.2 Performance metrics 37
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Grayscale byteplot and Markov plot malware visualizations</td>
<td>4</td>
</tr>
<tr>
<td>1.2</td>
<td>Byteplot with custom color mapping and RGB malware byteplot</td>
<td>5</td>
</tr>
<tr>
<td>1.3</td>
<td>bi-gram DCT plot</td>
<td>6</td>
</tr>
<tr>
<td>1.4</td>
<td>Visualization of op-code bi-grams and signal visualization</td>
<td>7</td>
</tr>
<tr>
<td>1.5</td>
<td>SOMs of Win NT apparition virus and its distribution</td>
<td>7</td>
</tr>
<tr>
<td>1.6</td>
<td>Audio-based visualization methods</td>
<td>8</td>
</tr>
<tr>
<td>1.7</td>
<td>Binary Visualization of Malware Klez: a,b,c,d</td>
<td>9</td>
</tr>
<tr>
<td>1.8</td>
<td>Behavior-based visualization</td>
<td>9</td>
</tr>
<tr>
<td>1.9</td>
<td>Visualization of the Netbull Virus Protected with the Mew Packer</td>
<td>10</td>
</tr>
<tr>
<td>1.10</td>
<td>Instruction frequency and Dotplot visualizations</td>
<td>10</td>
</tr>
<tr>
<td>2.1</td>
<td>RGB image</td>
<td>13</td>
</tr>
<tr>
<td>2.2</td>
<td>RGB color channels</td>
<td>14</td>
</tr>
<tr>
<td>2.3</td>
<td>Different interpretations of the same file</td>
<td>15</td>
</tr>
<tr>
<td>2.4</td>
<td>Different visualizations of the notepad.exe</td>
<td>16</td>
</tr>
<tr>
<td>3.1</td>
<td>AlexNet architecture</td>
<td>17</td>
</tr>
<tr>
<td>3.2</td>
<td>Convolution example</td>
<td>18</td>
</tr>
<tr>
<td>3.3</td>
<td>Half-padding example</td>
<td>19</td>
</tr>
<tr>
<td>3.4</td>
<td>Max-pooling example</td>
<td>20</td>
</tr>
<tr>
<td>3.5</td>
<td>Various activation functions</td>
<td>21</td>
</tr>
<tr>
<td>3.6</td>
<td>Fully connected layer example</td>
<td>21</td>
</tr>
<tr>
<td>3.7</td>
<td>Optimizer comparison</td>
<td>24</td>
</tr>
<tr>
<td>3.8</td>
<td>CNN example architecture</td>
<td>26</td>
</tr>
<tr>
<td>4.1</td>
<td>Sorel-20M S3 structure</td>
<td>30</td>
</tr>
<tr>
<td>4.2</td>
<td>Visualization of the SORRY dataset</td>
<td>32</td>
</tr>
<tr>
<td>4.3</td>
<td>CNN example architecture</td>
<td>33</td>
</tr>
<tr>
<td>4.4</td>
<td>Two visualized ransomware samples. Images from the SORRY dataset</td>
<td>33</td>
</tr>
<tr>
<td>4.5</td>
<td>Visualization of the Malevis dataset</td>
<td>34</td>
</tr>
<tr>
<td>5.1</td>
<td>Accuracy/loss graph</td>
<td>39</td>
</tr>
<tr>
<td>5.2</td>
<td>Confusion matrices for training and validation splits</td>
<td>39</td>
</tr>
<tr>
<td>5.3</td>
<td>Confusion matrix</td>
<td>40</td>
</tr>
<tr>
<td>5.4</td>
<td>Visualization of the CNN prediction</td>
<td>40</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Magic numbers and corresponding file types [48] ........................................ 14
2.2 Image width for various file sizes ............................................................. 15
3.1 List of popular cost functions [57] ............................................................. 22
4.1 Malware family distribution in SORRY dataset [19] ................................. 31
4.2 Malware family distribution in Malevis dataset [19] .................................. 35
5.1 Performance metrics comparison .............................................................. 41

List of code listings
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 particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 9, 2024
Abstract

Recently machine learning (ML) techniques become very popular in many areas, including natural language processing, voice/image recognition, etc. The goal of this master thesis is to create a technique of malware detection by using image visualization. As a method of gaining information from converted malware samples classification using Convolutional Neural Network (CNN) detection was chosen. The malware sample represented as byte array is first converted to gray-scale or RGB image and then fed as input to CNN.

Keywords  Malware detection, Malware visualisation, Convolutional Neural Network

Abstrakt

Techniky strojového učení (ML) jsou v poslední době velmi populární v mnoha oblastech, včetně zpracování přirozeného jazyka, rozpoznávání hlasu/obrazu atd. Cílem této diplomové práce je vytvořit techniku detekce malwaru pomocí vizualizace obrazu. Jako způsob získávání informací z převedených vzorků malwaru byla zvolena klasifikace pomocí detekce konvoluční neuronové sítě (CNN). Vzorek malwaru reprezentovaný jako bajtové pole je nejprve převeden na obrázek ve stupních šedi nebo RGB a poté převeden jako vstup do CNN.

Klíčová slova  Detekce malwaru, Vizualizace malwaru, Konvoluční neuronová síť
Abbreviations

ML  Machine Learning
CNN  Convolutional Neural Network
XGBoost  Extreme Gradient Boosting
SIFT  Scale-Invariant Feature Transform
AV  Antivirus
ORB  Oriented Fast and Rotated Brief
LBP  Local Binary Pattern
RGB  Red Green Blue
IoT  Internet of Things
PE  Portable Executable
VM  Virtual Machine
PC  Personal Computer
OP-code  Operational code
API  Application Programming Interface
GIST  Global Image Structure Tensor
PCA  Principal Component Analysis
DL  Deep Learning
ELM  Extreme Learning Machine
SVM  Support Vector Machine
DCT  Discrete Cosine Transformation
SARVAM  Search And RetriEVAl of Malware
SRC  Sparse Representation-based Classification
NN  Neural Network
RP  Random Projections
DNA  Deoxyribonucleic Acid
MIR  Music Information Retrieval
STFT  Short Time Fourier Transform
MIDI  Musical Instrument Digital Interface
WAV  Waveform Audio File Format
MFCC  Mel-frequency Cepstral Coefficients
HSV  Hue, Saturation, Value
YUV  luma, red projection and blue projection
L*a*b  Perceptual lightness, red, green, blue and yellow
JPEG  Joint Photographic Experts Group
PNG  Portable Network Graphics
TIFF  Tag Image File Format
GIF  Graphic Interchange Format
PDF  Portable Document Format
MP3  MPEG-1 Audio Layer-3
DOS  Disk Operating System
ELF  Executable Linkable Format
RAR  Roshal Archive
OS  Operating System
ReLU  Rectified Linear Unit
MSE  Mean Square Error
MAE  Mean Absolute Error
CCE  Categorical Cross Entropy
NLL  Negative Log Likelihood
Adagrad  Adaptive Gradient Algorithm
RMSProp  Root Mean Square Propagation
Adadelta  Adaptive Learning Rate Method
Adam  Adaptive Moment Estimation
SOREL-20M  Sophos-ReversingLabs 20 Million
SHA  Secure Hash Algorithm
FFNN  Feed Forward Neural Network
SORRY  SOREL Research imagerY
Introduction

Virus detection has been a big problem for decades. Humanity has come up with many different ways to detect malware, including signature detection and behavioral analysis. However, virus developers and hackers are coming up with increasingly sophisticated techniques to bypass existing detection methods.

But progress does not stand still. Since the invention of neural networks, they have increasingly entered our lives and appeared in various areas. Convolutional neural networks have performed well in image recognition and object detection. This is how the idea arose to combine the problem of malware detection and convolutional neural networks. Convolutional neural networks work well for grid-like structures. For example, an image can be represented as a tensor made up of pixels, where two dimensions are width and height, and the third is color depth. So, if we can turn a malicious file into an image, we can use it as an input to a convolutional neural network. By training it on a sufficient number of samples, it is possible to achieve high accuracy. This is what this thesis will be about.
Chapter 1

Research

In this chapter, I will analyze existing methods of malware detection and describe feature extraction processes they rely on.

We can look at malware from two main angles: static analysis and dynamic analysis. Each of them has its advantages and disadvantages. Static methods allow us to analyze malicious samples without running them. These methods work with the file content itself. They are fast and computationally undemanding. However, they cannot be that effective against criminals’ evasion techniques (e.g., packing or encryption). Dynamic methods don’t have this disadvantage, but they are much slower. For dynamic analysis, malware samples are usually run inside the virtual machine, and it takes some time to start it, run a sample, and then shut it down. In addition, malware writers also implement anti-debugging and anti-VM features when malware behaves differently on the real PC and inside the VM. Also, the action of the malware can be delayed by a couple of minutes or hours, leading to its malicious activity not being detected during the short time window during the analysis and false negatives.

Over time, malware detection methods have evolved. People have developed various features that can be extracted from malicious samples to improve detection accuracy. Some methods are based on byte sequences and byte n-grams ([1], [2], [3]), while others use a disassembler to obtain OP-codes and use OP-code sequences or OP-code n-grams for detection and classification ([1], [5]). Using a control flow graph [6], API call sequences ([7], [8], [9]) or printable strings [10] are other popular methods. In this thesis, I will study non-standard way of detecting malware, namely using visualization.

1.1 Related work

The idea of representing malicious files as images and using this representation for further detection or classification isn’t quite new. Many articles describe different ways to achieve this.

One of the first articles was published in 2011 by Nataraj et al. [11]. The method proposed here looks at the malware samples as a byte sequence, where each byte is then turned into a pixel value of the grayscale image (1.1a). Received images were used to create the MalImg dataset. The authors extracted 320 GIST features from the images and applied the k-nearest neighbor algorithm for malware classification. This approach was then compared to dynamic analysis in [12] and showed comparable results with much less computational and time costs. In [13], Narayanan et al., in a similar way, converted malware samples into images of size $256 \times 16$ pixels and then used principal component analysis (PCA) for the feature extraction. The article shows that the best accuracy of 96.6% was achieved using 12 PCA features and a linear k-nn classifier. Research [14] compares different classification methods based on various malware
features, including N-gram, metadata, entropy, etc. Among other things, the authors used grayscale byteplot images and extracted Harralick and Local Binary Pattern features from them. These methods showed 95.5% and 95.1% accuracy, respectively, using the XGBoost classifier. In [15], in addition to using grayscale byteplot, authors also use a visualization method called Markov plot [15b]. They also calculated different keypoint descriptors (SIFT, ORB) and extracted different texture features (LBP, Local Entropy, Gabor Filters) from obtained images. They also tested extracted features in combination with various ML classifiers. The combination of grayscale byteplot, Gabor Filters, and Random Forest classifier showed the best F1-score, which was 0.9948 and even outperformed several solutions from AV companies.

Despite being faster than dynamic analysis, an extraction of additional features still requires additional image processing and calculation. To avoid these calculations, multiple authors resorted to using Deep Learning (DL) in their papers. Gibert et al., in their research [16], designed their convolutional neural network and compared their results with the state-of-the-art methods described in [11], [13], [14]. It turned out that the CNN architecture proposed in [16] outperformed all these methods. In [17], the authors also investigate whether the number of GIST features proposed in [11] can be reduced and what number is optimal to show similar accuracy. In the second part of this article, they used the Inception network created by Google on raw image bytes without any preprocessing and achieved good results. The same visualization approach was used in [18]. Here, the authors proposed simple 1- and 2-layer CNN architecture trained on raw pixel values and compared it to ELM, which trained on the same data. It turned out that ELM can achieve an equal or better F1 score with only 2% of the time needed to train comparable CNN. In [19], the author used a similar visualization approach, but instead of 1-1 mapping of malware bytes to grayscale image pixels, they converted malware into a 3-channel RGB image (three consecutive bytes of malware are mapped to one image pixel) [15]. To make this conversion, they used bin2png python script developed by Sultanik, publicly available on GitHub [20]. This is how the Malevis dataset arose. After that, they used the received dataset and tested how different popular CNN architectures would behave in terms of malware classification.

![Grayscale byteplot and Markov plot malware visualizations](image)

**Figure 1.1** Grayscale byteplot and Markov plot malware visualizations
As we can see, convolutional neural networks are widely used in malware classification. However, it can be challenging to design suitable CNN architecture. In [21] authors converted malware samples into greyscale images and applied custom color mapping to it Fig. 1.2a. As a model, they used modified VGG-16 CNN and then applied fine-tuning to reach better accuracy. It is interesting that colored images showed slightly better accuracy - 98.82% vs. 98.27%. Kalash et al. [22] also used VGG-16 for malware classification in their work. The achieved accuracy was 98.52% for MalImg [11] and 98.99% for Microsoft Malware Dataset. A similar approach was proposed by Sudhakar and Kumar in [23]. They used modified ResNet50 and transfer learning to solve malware classification problem. The best accuracy was achieved in combination with Adam optimizer and amounted to 99.1862%. Authors of [24] used a genetic algorithm to find CNN architecture automatically. Obtained CNN showed lower accuracy than the methods proposed in [21], [23], and [22], but the number of CNN parameters was also lower. The smallest CNN proposed in the article has only 83,433 parameters, compared to 23,552,969 in [23]. That makes it suitable for small-capacity devices and IoT.

The interesting approach was proposed in [25], where authors used pre-trained CNN models (such as VGG-16, Xception, ResNet152) only for extracting features from malware images. These features were later passed on as input to other classifiers. The best accuracy of 99.3% was achieved by the combination of the VGG-16 network and random forest algorithm.

Bensaoud et al. [26] compared multiple classifiers for malware classification and detection, including CNN-SVM, where the softmax layer was replaced with a linear SVM. The experiment showed that Inception-v3 reached the best accuracy compared to other models both in classification and detection tasks with 99.25% and 99.24% accordingly. CNN-SVM achieved only 93.22% accuracy in classification but showed 99.11% accuracy in malware detection, which is the second-best result.

In [27], authors propose a new visualization technique where they calculate the bi-gram frequency of a malware sample. The obtained values are then turned into a 256x256 image pixel value. Finally, they apply Discrete Cosine Transformation (DCT) to get a resulting image pattern Fig. 1.3. For malware detection, the authors propose a custom shallow CNN architecture consisting of 3 convolutional and two density layers (3C2D) and compare its performance to a fine-tuned ResNet50 model. As the input to CNN, they combined standard grayscale byteplot with bi-gram DCT. The deep CNN model (ResNet) performed much better than the k-nn, and
the RF classifier showed slightly better results than the proposed 3C2D architecture and achieved 96% mean binary accuracy.

Authors of [28] adapted code from publicly accessible GitHub repositories [29] and [30] and extracted 54 non-image features from malware binaries. After that, they utilized the k-nn classifier together with these features and compared its performance with ResNet34 + malware byteplot visualization. The experiments showed that a much simpler k-nn classifier outperformed CNN in some cases. However, CNN showed better results in zero-day attack detection, which may indicate that convolutional neural networks can better generalize training data.

To improve prediction accuracy, authors of [31] proposed to combine multiple CNNs into an ensemble. They used pre-trained VGG-16 and ResNet50 for feature extraction, then applied PCA to reduce the number of features and passed these vectors to One-vs-All Multiclass SVMs. In parallel with this, the images were fed to the input of the fine-tuned VGG-16 and ResNet50 with softmax classifier. The results of all predictions were then gathered, and the image family was obtained by integrating posterior probability. The proposed method reached 99.5% accuracy on the MalImg dataset [11] and outperformed all other methods.

In [32], authors invented a method of getting pixel values from op-code sequences. They disassembled the malware binary and then grouped received op-codes into groups of 2. After that, they calculated the pixel value based on the current group probability and the information gain. The result after histogram normalization is shown on the 1.4a. The best accuracy was achieved by utilizing the ResNet152 model and was 88.3%.

In [33], authors for their experiments again used grayscale byteplot, but they also showed another visualization method, where the X-axis is the byte number of malware sample and the Y-axis represents byte value in the range [0:255]. The first part of the article was focused on developing an online system for large-scale malware search and retrieval called SARVAM [34]. For this, the authors extracted GIST features and used Balltree-based NN for faster search. For a new query, it takes around six seconds to find a match. The second part of the article is dedicated to the sparse representation-based classification (SRC) framework. They represent an unknown malware as a sparse linear combination of the training samples and then use Random
Related work

Projections (RP) to decrease the vector’s dimension. To estimate the family of malware, they compute residuals for every family in the training set and then select the family with the minimal residue. The highest accuracy was obtained for the combination of RPs and the SRC framework and reached 92.83% for Malimg and 98.55% for Malheur dataset [33].

![Figure 1.4](image1.png) **Figure 1.4** Visualization of op-code bi-grams and signal visualization

In addition to the above methods, there are a lot of other interesting and non-standard methods of malware visualization. In [35], the author used self-organized maps for detecting and localizing malware impact at Windows PE executables. They showed that each virus has its unique virus mask, like DNA, which can be used to determine malware family 1.5.

![Figure 1.5](image2.png) **Figure 1.5** SOMs of Win NT apparition virus and its distribution [35]

Another way to identify malicious software is to convert the malware executable into an audio file. It then allows us to apply different Music Information Retrieval (MIR) techniques for further feature extraction. In [36], authors convert malware samples into audio signals and use Short Time Fourier Transform (STFT) to calculate spectrograms 1.6a. For the classification, they used three CNN architectures with 1, 2, and 3 convolution layers, respectively, and compared their performances. The experiment shows that the best accuracy, 92.8%, was achieved by a deeper CNN architecture. The authors also compared spectrogram visualization with standard
grayscale byteplot. It turned out that the spectrogram approach showed better results in both malware detection (92.8% vs 92.3%) and malware classification (96% vs 95.5%). Farrokhmanesh and Hamzeh [37] made a MIDI file out of a malware sample and used a MIDI synthesizer to create a WAV file. Then, they extracted the most common audio features, such as MFCCs and chromagrams. Obtained feature vectors were later used to train ML classifiers. In the experiment, AdaBoost outperformed k-nn and Random Forest classifiers and showed 92.2% accuracy. Audio signal processing for Android malware detection was studied in [38]. As in the previous paper, authors converted malware executables into WAV audio files and extracted multiple audio features, including chromagram, MFCC, RMS, Spectral Centroid, and others. They also compared the efficiency of different ML classifiers. The best result was achieved by the NN classifier and was 95.2% for the malware detection task and 92.2% for the family identification one. Similar approaches were used in [39] and [40], where authors combined visual, audio, hash, and structural analysis to detect malicious samples. In [39] used grayscale byteplot, bi-gram DCT image representation together with MFCCs, melspectrogram, and chromagram extracted from the audio representation of a malware. They also calculated the ssdeep hash of a sample and created a bag-of-words model. After numerous experiments, it turned out that the best accuracy of 99.92% was shown by combining bigram-dct, MFCC, and bag-of-words features. In [40], authors research different methods of feature extraction as well as their orthogonality. It can help to find out which feature sets complement each other and can lead to an increase in accuracy. Among features extracted from audio files, melspectrogram and extended MFCC [40] showed the best performance. The paper also showed that audio and image features are the most orthogonal to pehash and instruction count. So it would be a good idea to combine them. Sharma and Raglin [41] in their article transformed malware samples to WAV audio format and computed MFCC descriptors. They also developed their analysis method called Entropy Density based on Shannon entropy [41].

Authors of article [42] disassembled malware sample to extract several metrics. The metrics are the number of control transfer instructions, the total number of instructions within a function, and the number of data transfer instructions. They assigned each metric its axis on the graph, which allowed them to visualize malware signature in three-dimensional space. We can see that malware belonging to the same family has a very similar 3D signature.
utilizes dynamic analysis to get information about malware behavior. The behavioral report obtained from CWSandbox is then transformed into two different views: treemap and thread graph. The first one looks like a set of nested rectangles; each one is related to an API calls group. The width of the rectangle depends on the number of such calls. The thread graph view shows the development of malware activity. Here, the X-axis represents the time, and the Y-axis represents the operation performed. This can be considered as a behavioral fingerprint of a sample and allows the analyst to better understand the actions of individual threads.

In [44], authors used the monitoring framework Ether to track and analyze malware behavior.
They collected program instructions and visualized them as a directed graph of basic blocks. The authors also came up with custom color coding, which helps visually distinguish between packed and normal instructions and detect dynamically generated or self-modifying code. They also created a tool with a Google Maps alike interface, when it is possible to interact with the graph.

Figure 1.9 Visualization of the Netbull Virus Protected with the Mew Packer

In [45], authors used cluster ensemble - a method of getting consensus between different clustering methods and used it to aggregate a solution obtained by partitional and hierarchical clustering methods. They chose instruction frequency and function-based instruction sequences as features for extracting from malware samples. At the 1.10a, we can see that instruction frequency patterns are shared between the same malware families and differ between different families.

Figure 1.10 Instruction frequency and Dotplot visualizations

In paper [46], authors extract malware sections from its memory representation to avoid packing and use the Dotplot visualization technique to find similarities between them. Authors use n-gram to avoid irrelevant matches. High-level (functional) similarity occurs less often than similarity on a byte level and is more important. When 16-gram of the first malware matches 16-gram of the second malware, a black dot is drawn; otherwise, a white dot is drawn. After
that, hash sampling was applied to reduce the overall Dotplot size. The final result is shown at [1,10].
This chapter covers the theory of transforming the malware executable file into an image, which can later be used for model training and malware detection.

### 2.1 Image format

Storing an image digitally can be straightforward, where each pixel is saved using either one or three bytes, depending on the number of color channels. Nevertheless, this method is highly inefficient and can result in excessively large files. For instance, an image captured by a typical smartphone might require over 29 megabytes of storage. Therefore, images are typically compressed using various formats, significantly reducing the file size while degrading quality almost imperceptibly to the human eye. [47]

A bitmap image stores an image with one byte per pixel for grayscale images and three bytes per pixel for color images. The best-known color scheme is RGB [27] The separate channels are shown at [2.2] Notice that the red parts are lighter on the red layer, the blue parts on the blue layer, and the green parts on the green layer. At the same time, other colors on each layer are darkened. The white pixels are bright on all layers. [47]
This format does not use compression and usually has extensions like .bmp, .tga, .pgm, and .ppm. There are other color schemes (HSV, YUV, L*a*b) and image formats (JPEG, PNG, TIFF, etc), but once we have the image in one of the formats we can easily convert them between each other.

2.2 Malware to image conversion

On a computer, all files are stored in binary form. A bit is the smallest unit of information in computer systems. But operating with individual bits is usually inconvenient and unnecessary. Most often, when describing something, we talk about bytes.

All of us are familiar with different file formats, such as images, text, videos, executable files, etc. But in reality, a file is just a sequence of bytes, and its type depends on how we interpret this sequence. The software relies on identifiers known as magic numbers, a byte sequence typically located at the beginning of a file. They are unique to each file format, and thanks to them, the computer determines how to interpret the file. For example:

<table>
<thead>
<tr>
<th>Hex Signature</th>
<th>File Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>89 50 4E 47 0D 0A 1A 0A FF D8 FF E0</td>
<td>PNG Image</td>
</tr>
<tr>
<td>47 49 46 38 37 61</td>
<td>JPEG Image</td>
</tr>
<tr>
<td>25 50 44 46 2D</td>
<td>GIF Image</td>
</tr>
<tr>
<td>52 49 46 46 ?? ?? ?? ?? 57 41 56 45 49 44 33 4D 5A</td>
<td>PDF Document</td>
</tr>
<tr>
<td>7F 45 4C 46</td>
<td>WAV Audio</td>
</tr>
<tr>
<td>50 4B 03 04</td>
<td>MP3 Audio</td>
</tr>
<tr>
<td>25 21 50 53</td>
<td>DOS MZ executable</td>
</tr>
<tr>
<td>52 61 72 21 1A 07 00</td>
<td>ELF Executable</td>
</tr>
<tr>
<td>21 25 50 53</td>
<td>ZIP Archive</td>
</tr>
<tr>
<td>52 61 72 21 1A 07 00</td>
<td>PostScript Document</td>
</tr>
<tr>
<td>52 61 72 21 1A 07 00</td>
<td>RAR Archive</td>
</tr>
</tbody>
</table>

But we don’t need to follow these recommendations and can, e.g., open images or exe as text. Usually, we will see something incomprehensible, but sometimes it can be useful, for example, for steganography. Here you can see the same file opened as image and as text.
We already know that in a one-channel grayscale image, each pixel consists of one byte. So, the most simple way to convert a file into an image is to map each byte from the source file to one byte in the resulting image. This approach was successfully applied by Nataraj et al. in [11]. The image width is fixed according to the table 2.2 but the height can vary depending on the file size:

<table>
<thead>
<tr>
<th>File size range</th>
<th>Image width</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 10 kB</td>
<td>32</td>
</tr>
<tr>
<td>10 kB - 30 kB</td>
<td>64</td>
</tr>
<tr>
<td>30 kB - 60 kB</td>
<td>128</td>
</tr>
<tr>
<td>60 kB - 100 kB</td>
<td>256</td>
</tr>
<tr>
<td>100 kB - 200 kB</td>
<td>384</td>
</tr>
<tr>
<td>200 kB - 500 kB</td>
<td>512</td>
</tr>
<tr>
<td>500 kB - 1000 kB</td>
<td>768</td>
</tr>
<tr>
<td>&gt; 1000 kB</td>
<td>1024</td>
</tr>
</tbody>
</table>

A slightly improved way is to use not one channel but three channels, sequentially reading bytes from the file. This way, we get an RGB image. In this case, each pixel of the output image will consist of three bytes of the input file: the first byte will be encoded in the red channel, the second byte in the green channel, and the third byte in the blue channel. If the file size is not the square of the number multiplied by the number of channels, then zero padding is applied. A similar algorithm was used by Bozkir et al. in the process of creating the Malevis dataset [19]. As we can see, this conversion is file and OS-agnostic, as we are working with raw bytes instead of OS/extension specifics. Such conversion gives us the ability to easily manipulate malware samples by using image libraries and applying various image transformations. Our assumption is that different malware families will have different visual patterns, and based on this, we will be able to easily detect them.

Here is my vision of the conversion algorithm:
Algorithm 1: Binary to image conversion

Input: Input file path `file_path`
Input: Number of channels `num_channels`

Read all bytes from the `file_path` into `data` and store the file size as `data_size`;

Calculate the dimensions of the output image: \( \text{width} \leftarrow \lceil \sqrt{\lceil \frac{\text{data\_size}}{\text{num\_channels}} \rceil} \rceil \);

Calculate the expected output size: \( \text{output\_size} \leftarrow \text{num\_channels} \times \text{width}^2 \);

Append \((\text{output\_size} - \text{data\_size})\) zeros to `data`;

for \( \text{row} = 0 \) to \( \text{width} - 1 \) do
  for \( \text{col} = 0 \) to \( \text{width} - 1 \) do
    for \( \text{ch} = 0 \) to \( \text{num\_channels} \) do
      \( \text{pixel\_index} \leftarrow \text{row} \times \text{width} \times \text{num\_channels} + \text{col} \times \text{num\_channels} + \text{ch} \);
      \( \text{output\_row, col, ch} \leftarrow \text{data}_{\text{pixel\_index}} \);
  
Save the output as an image file;

This code could be optimized by using function like `np.reshape` instead of nested loops, but I implemented it "as is". Below is the result of the algorithm: grayscale and RGB version of the `notepad.exe`.

(a) Grayscale

(b) RGB

**Figure 2.4** Different visualizations of the notepad.exe
Chapter 3

Convolutional Neural Networks

With the development of deep learning technologies, convolutional neural networks (also known as CNN) have become very popular. They have proved their efficiency in processing grid-like data. That could be anything from a time-series 1-D array representing samples taken at regular time intervals to applications where CNN is used for natural language processing. However, the most widespread use of this neural network is image processing. As an image can be represented as a grid of pixels of some dimension, CNN can be used to perform such operations as image classification, object detection, image segmentation, etc. [50]. The idea of the convolutional neural network refers us back to biology as it was inspired by the workings of the cat’s visual cortex [51]. Hubel and Wiesel discovered that a cat’s cortical neurons most strongly react to specific light patterns, such as oriented bars, and hardly react to others. Also, individual neurons are activated only by a small part of the entire visual field, known as a receptive field. The receptive fields of various neurons overlap to some extent, ensuring complete coverage of the entire visual field [51].

3.1 Building blocks

A convolutional neural network belongs to the class of deep learning neural networks, and its depth can reach tens of layers connected to each other. For example, the ImageNet Large Scale Visual Recognition Challenge 2012 winner CNN called AlexNet is eight layers deep [52], has 650,000 neurons and 60 million trainable parameters [52]. Its architecture can be seen at 3.1.

Figure 3.1 AlexNet architecture [53]
In addition, there are several types of layers, each of which performs its own specific operation. In this section, I will describe what convolution is, as well as the other layers of a CNN and the mathematics behind it.

3.1.1 Convolution

In a convolutional neural network, the state of each layer is represented by a spatial grid structure. The values and shape of the first layer are determined by the nature of the input data. It is called the input layer. Parameters that we want to obtain during training in CNN are organized into 3-dimensional structures, also known as kernels or filters. The filter usually has the same width and height (is a square), and its depth is always the same as the depth of the layer to which it is applied. By sequentially applying a filter to the input layer, we will get more layers. These later newly obtained layers are referred to as hidden layers, and their grids are referred to as feature maps or activation maps. Here, "applying filter" means placing the filter at each possible position in the input (or hidden) layer so that it stays fully within its borders and then performing a dot product between the filter and the matching grid in the input. This operation is called convolution.

![Figure 3.2](image)

**Figure 3.2** An example of a convolution between a $3 \times 4 \times 1$ input and a $2 \times 2 \times 1$ filter

**Definition 3.1** (Convolution). Let $W^{(p,q)} = [w^{(p,q)}_{i,j,k}]$ represent a 3-dimensional tensor of size $F_q \times F_q \times d_q$, which contains the parameters of the $p$th filter in the $q$th layer. Here, indices $i$, $j$, and $k$ denote the positions along the height, width, and depth of the filter, respectively. Additionally, the 3-dimensional tensor $H^{(q)} = [h^{(q)}_{i,j,k}]$ with dimensions $L_q \times B_q \times d_q$ represents the feature maps in the $q$th layer. If $q = 1$, $H^1$ simply corresponds to the input layer. Then, the convolutional operations from the $q$th layer to the $(q + 1)$th layer can be defined as follows:
Building blocks

\[ h_{i,j,p}^{(q+1)} = \sum_{r=1}^{F_q} \sum_{s=1}^{F_q} \sum_{k=1}^{d_q} w_{r,s,k}^{(p,q)} h_{i+r-1,j+s-1,k}^{(q)} \forall i \in \{1, \ldots, L_q - F_q + 1\} \]
\[ \forall j \in \{1, \ldots, B_q - F_q + 1\} \]
\[ \forall p \in \{1, \ldots, d_{q+1}\} \]

The resulting dimensions of the new layer will be \( L_{q+1} \times B_{q+1} \times d_{q+1} \), where \( L_{q+1} = (L_q - F_q + 1) \), \( B_{q+1} = (B_q - F_q + 1) \) and \( d_{q+1} \) is equal to the number of filters used in \( q \)th layer. [53]

In subsequent sections, already-defined notations will be used without redefinition.

### 3.1.2 Padding

The operation described in the previous section is usually referred to as “valid convolution” because the filter is moving within the borders of the layer, without “sticking out”. Experiments show that it generally does not work well. The number of positions in which the filter captures outer pixels is less than the number of pixels that are closer to the center. This way, the contribution of the pixels around the border will be under-represented in the final result. To avoid this, the original size of the layer can be extended by zeros on each side. This operation is called padding. There are several types of padding. [53]

No padding, sometimes also called valid-padding, is when the input is left unchanged, meaning no zeros are added to the sides. Another extreme is called full-padding, where the input is padded with \((F_q - 1)\) zeros around the borders. This is equivalent to allowing the filter to almost fully “stick out”. The resulting dimension will be \( L_q + 2(F_q - 1) \times B_q + 2(F_q - 1) \). And finally, the most popular one is half-padding. [3.3] The main property of half-padding is that it preserves feature map size after convolution. During convolution, height, and width are both reduced by \((F_q - 1)\). So, if we pad the input with \((F_q - 1)/2\) zeros on each side before performing convolution, the special footprint will remain the same. [53]

![Figure 3.3](image.png)

**Figure 3.3** An example of half-padding applied to \(7 \times 7\) feature map

### 3.1.3 Strides

In the previous definition of convolution, the filter always shifted by 1. Nevertheless, it can be taken as another input parameter to convolution. Such a parameter is usually referred to as a stride. When a stride of \(S_q\) is used, the kernel is shifted by \(S_q\) locations during convolution. In such case the size of output layer has height \((L_q - F_q)/S_q + 1\) and a width of \((B_q - F_q)/S_q + 1\). High stride value can help to reduce memory usage and over-fitting; however, in most cases, a
stride of 1 or 2 is used. Increasing the strides can quickly expand the receptive field of individual features while reducing the spatial footprint of the entire layer. [53]

3.1.4 Pooling

Pooling is another important operation in CNN, which yields something called translation invariance. This property ensures that small changes to the input will not significantly change the result. It is especially useful in image recognition. Even if an object is slightly shifted in the original image, it can still be identified as the same object. [53]

Unlike convolution, pooling is applied separately on each feature map. Let’s take a region of size $P_q \times P_q$. Depending on the type, pooling ”squeezes” all values of the input within this region into a single value. In the case of max-pooling, the result is the maximum element in the region, while for average pooling, it’s the mean value. Then, the region is shifted by some stride (usually 1 or 2), and the operation is repeated for all possible positions within the borders [53]. The resulting size will be $\left( L_q - P_q \right)/S_q + 1 \times \left( B_q - P_q \right)/S_q + 1$.

\[ \text{Figure 3.4} \text{ An example of } 3 \times 3 \text{ max-pooling applied to } 7 \times 7 \text{ feature map with strides 1 and 2} \]

3.1.5 Activation function

The activation function in neural networks acts as a threshold and activates the neurons that contribute to the prediction the most. It is worth noting that non-linear activation functions are usually used. They allow us to capture the nonlinearity of the data and better generalize the dataset.

In convolutional neural networks, the activation layer usually follows the convolution layer and often is not explicitly shown in pictures with CNN architecture. The activation function is applied to each of the $L_q \times B_q \times d_q$ values of the feature map, and the result of the operation is $L_q \times B_q \times d_q$ thresholded values. [53]

There’s a variety of activation functions, ranging from simple ones like identity and sign to more complex ones like sigmoid, tanh, and hard tanh [3.7]. The most used activation function in CNN is called the Rectified Linear Unit function (ReLU), defined by:
Building blocks

\[ f(x) = \begin{cases} 
0, & x < 0 \\
x, & x \geq 0 
\end{cases} \]  \hfill (3.1)

![Various activation functions](image)

**Figure 3.5** Various activation functions [53]

Earlier, functions such as tanh or sigmoid were also used to train CNN, but it appeared that ReLU surpasses them both in speed and accuracy. [54]

### 3.1.6 Fully connected layer

All feature maps of the last convolutional layer of size \( L_q \times B_q \times d_q \) are flattened into one vector of \( L_q \times B_q \times d_q \) elements and fed as the input to the **fully connected layer**, which is nothing more than a feed-forward network. As its layers are densely connected, a fully connected layer is also called **dense layer**. Because of connection density, most of the parameters lie in these layers. E.g., if there are two fully connected layers with 4096 neurons each, then there will be more than 16 million connections between them. Typically, several such layers are added in order to increase the computing power of the network. The output of the neural network then depends on the specific application. One might select the logistic, softmax, or linear activation function depending on whether the goal is classification or regression. [53]

![An example of the fully connected layer with three neurons](image)

**Figure 3.6** An example of the fully connected layer with three neurons.
For the classification, the most popular choice is softmax. It converts a vector of arbitrary real-valued scores (often called logits) into a vector of probabilities, ensuring that the probabilities sum up to 1. \[\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}}\] (3.2)

The resulting class is then the class with the maximal probability.

### 3.2 Training

Now that we know the functions of the individual layers and have put them together into some kind of architecture, we need to understand how to find the correct values for each of the parameters. We can try to find a successful combination ourselves or set the values randomly, but we are unlikely to get high accuracy in the end. We can also show the neural network the data along with the correct answers and let it find the most appropriate parameters. The process of learning these values is called training. In this section, I will describe how the training is done in convolutional neural networks and beyond.

#### 3.2.1 Loss function

For the deep learning algorithms, we need some measure of error to tell us how far our prediction is from the correct one. In practice, training comes down to minimizing some function \(f(x)\) by altering \(x\). This function is usually referred to as the cost function, loss function, or the error function. \[x^* = \arg\min f(x)\] (3.3)

Different loss functions are used for different problems. In regression tasks, the model predicts an output value corresponding to an input value rather than pre-selected labels. On the other hand, for the classification model produces a vector of probabilities indicating the input’s likelihood of belonging to various pre-defined categories. The category with the highest probability is then selected. Below is the table of the most popular cost functions and their applications:

<table>
<thead>
<tr>
<th>Loss function</th>
<th>Regression</th>
<th>Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean Square Error (MSE) / L2 Loss</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Mean Absolute Error (MAE) / L1 Loss</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Huber Loss / Smooth Mean Absolute Error</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Binary Cross-Entropy Loss</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Categorical Cross-Entropy Loss</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Hinge Loss</td>
<td>×</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Table 3.1** List of popular cost functions

Certainly, numerous other cost functions exist, and custom cost functions can also be created if needed. The most used loss function for CNN training is categorical cross-entropy (CCE), also called negative log-likelihood (NLL).

▶ **Definition 3.2** (Categorical cross-entropy). Let \(N\) represent the number of examples in the dataset, \(K\) be the number of classes, and \(\theta\) represent the parameters of the model. If \(y_k^{(i)}\) is
a binary indicator \((0 \text{ or } 1)\) indicating whether class \(k\) is the correct classification for the \(i\)-th example, and \(\hat{y}_k^{(i)}\) is the predicted probability that the \(i\)-th example belongs to class \(k\), then the categorical cross-entropy loss function is defined as:

\[
J(\theta) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{k=1}^{K} y_k^{(i)} \log(\hat{y}_k^{(i)})
\]

In the future, this will be a function that we want to minimize. \(^5\)

### 3.2.2 Gradient descent

In machine learning, we usually aim to minimize functions with multiple inputs: \(f : \mathbb{R}^n \to \mathbb{R}\). In order to achieve this we need to use the concept of partial derivatives. The partial derivative \(\frac{\partial f}{\partial x_i}\) measures how the function \(f\) changes when only the variable \(x_i\) increases at point \(x\). The gradient \(\nabla_x f(x)\) is a vector, where \(i\)-th element of the gradient is the partial derivative of \(f\) with respect to \(x_i\). \(^5\)

\[
\nabla_x f(x) = \left( \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right) \quad (3.4)
\]

In higher dimensions, critical points are locations where each component of the gradient equals zero. For some functions, it is possible to jump directly to the critical point by solving the equations \(\nabla_x f(x) = 0\), but in deep learning, where millions of parameters are involved, it is almost impossible. So, instead, the iterative method is used. We utilize the fact that moving in the direction opposite to the gradient allows us to reduce the value of \(f\). This approach is commonly referred to as the method of **steepest descent** or **gradient descent** \(^5\). We update the current parameter vector \(x\) by subtracting a fraction of the gradient of the function \(f\) with respect to \(x\). This is performed according to the following equation:

\[
x' = x - \eta \nabla_x f(x) \quad (3.5)
\]

Here, \(\eta\) denotes the learning rate, which is a positive scalar that determines the size of the step \(^5\). One common approach is to set \(\eta\) to a small constant. However, it may be difficult to find appropriate value. That’s why algorithms with adaptive learning rates were invented. These include such algorithms as Adagrad, RMSProp, Adadelta, Adam, and others \(^5\). The reader can learn more about how these algorithms work in the books \(^5\) or \(^3\), and below, I will describe only one of them, namely Adam.

The name “Adam” originates from the term "adaptive moments". It can be considered as a variation that combines RMSProp and momentum but with several key differences. First, momentum is directly incorporated by estimating the first-order moment of the gradient using exponential weighting. Second, it also deals with addresses the bias inherent in exponential smoothing when the estimate of a smoothed value is unrealistically set to 0. \(^5\) \(^3\)

Just like other algorithms, Adam uses something called **minibatching**. Typically, the dataset is divided into two parts: **training set** in order to train the model itself and **testing set** to evaluate the results. As the dataset size increases, the time to calculate one gradient step becomes prohibitively long. Instead, we can approximate it by using only a small set of samples. In every iteration of the algorithm, we randomly select a minibatch of samples, denoted as \(B = x^{(1)}, \ldots, x^{(m)}\) from the training set. The size of the minibatch is typically kept relatively small, ranging from just one to a few hundred examples, and doesn’t change with the dataset size. By using this approach, it is possible to fit a training set with billions of examples using steps computed on only hundreds of them. \(^5\)
Algorithm 2: The Adam algorithm \[55\]

**Input:** Step size $\epsilon$ (Suggested default: 0.001)

**Input:** Exponential decay rates for moment estimates, $\rho_1$ and $\rho_2$ in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)

**Input:** Small constant $\delta$ used for numerical stabilization. (Suggested default: $10^{-8}$)

**Input:** Initial parameters $\theta$

Initialize 1st and 2nd moment variables $s = 0$, $r = 0$;

Initialize time step $t = 0$;

while stopping criterion not met do

Sample a minibatch of $m$ examples from the training set $x^{(1)}, \ldots, x^{(m)}$ with corresponding targets $y^{(1)}$;

Compute gradient: $g \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(x^{(i)}; \theta), y^{(i)})$;

$t \leftarrow t + 1$;

Update biased first moment estimate: $s \leftarrow \rho_1 s + (1 - \rho_1)g$;

Update biased second moment estimate: $r \leftarrow \rho_2 r + (1 - \rho_2)g \odot g$;

Correct bias in first moment: $\hat{s} \leftarrow \frac{s}{1 - \rho_1^t}$;

Correct bias in second moment: $\hat{r} \leftarrow \frac{r}{1 - \rho_2^t}$;

Compute update: $\Delta \theta = -\sqrt{\hat{s}} \odot \sqrt{\hat{r}} + \delta$ (operations applied element-wise);

Apply update: $\theta \leftarrow \theta + \Delta \theta$;

Adam has shown empirical effectiveness, making it a popular choice for deep learning tasks in recent years. Below, you can see the visualization of the behavior of many optimizers: gradient descent (cyan), momentum (magenta), Adagrad (white), RMSProp (green), and Adam (blue) on different surfaces. As you can see, for several setups, Adam was able to find the global minimum, while others got stuck at the local minimum.

\[ \text{Figure 3.7 Comparison of gradient descent, momentum, Adagrad, RMSProp, and Adam} \]
The visualization was created by me using a tool developed by Lili Jiang \cite{58}.

### 3.2.3 Dropout

Dropout is a regularization method that randomly sets the output of a neuron in a non-output layer to zero with a certain probability $p$. \cite{54} As a result, we obtain numerous new network architectures. Using multiple models to make decisions is referred to as ensemble learning. Unlike bagging, where we train each model separately on different data, dropout allows us to take advantage of weight sharing. \cite{55} It also helps avoid co-adaptation, making neurons learn features independently of each other. This reduces the risk of overfitting and increases robustness as it forces a certain level of redundancy. During testing, neurons remain untouched and are not dropped out. \cite{55}

Due to direct dropout, after training, the outputs of the neurons will be larger than usual. Therefore, during the testing, all outputs must be multiplied by keep probability $(1 - p)$, which involves network modification. Popular deep learning frameworks implement something called inverted dropout. In this approach, activations are scaled by the inverse of keep probability $\frac{1}{1 - p}$ during the training phase while keeping the test phase unchanged \cite{59}.

### 3.2.4 Cross-validation

Sometimes, it can be challenging to split a dataset into testing and training sets due to a limited number of samples. A small test set introduces statistical uncertainty around the estimated average test error. This is where the idea of repeating the training and testing process on different randomly chosen subsets of the original dataset comes in handy. One commonly used technique for this purpose is \textit{k-fold cross-validation} \cite{55}. In \textit{k-fold cross-validation}, new datasets are generated by dividing the original dataset into \textit{k} non-overlapping subsets. The test error is then estimated by averaging the test error across \textit{k} trials. During each trial $i$, the $i$-th subset of the data serves as the test set, while the remaining data are used as the training set. \cite{55}

\begin{algorithm}
\caption{3.2.4 Cross-validation \cite{55}}
\begin{algorithmic}
\State \textbf{Input:} $D$, the given dataset, with elements $z^{(1)}$
\State \textbf{Input:} $A$, the learning algorithm, seen as a function that takes a dataset as input and outputs a learned function
\State \textbf{Input:} $L$, the loss function, seen as a function from a learned function $f$ and an example $z^{(i)} \in D$ to a scalar $\in \mathbb{R}$
\State \textbf{Input:} $k$, the number of folds
\State Split $D$ into $k$ mutually exclusive subsets $D_i$, whose union is $D$;
\For {$i = 1 \text{ to } k$}
\State $f_i = A(D \setminus D_i)$;
\For {$z^{(j)} \text{ in } D_i$}
\State $e^{(j)} = L(f_i, z^{(j)})$;
\EndFor
\EndFor
\Return $e$;
\end{algorithmic}
\end{algorithm}

### 3.2.5 Backpropagation in CNN

Once we have learned the functions of the individual layers, we need to understand how the error is propagated back in order to update the weights. Let’s take as an example the following convolutional neural network architecture consisting of a convolutional layer, ReLU, average pooling, another convolutional layer, ReLU, max-pooling, and a fully connected layer, where $C^{[1]}$, $Z^{[1]}$, $P^{[1]}$, $C^{[2]}$, $Z^{[2]}$, $P^{[2]}$ and $y$ are respective results.
To update the weights we need to calculate \( \frac{\partial L}{\partial K} \), \( \frac{\partial L}{\partial K} \), \( \frac{\partial L}{\partial W} \), and \( \frac{\partial L}{\partial B} \). Let’s start from the fully connected layer. Forward propagation in the fully connected layer is defined by the following equations:

\[
\begin{align*}
  f &= \text{flatten}(P^{[2]}) \\
  Z^{[3]} &= W^{[3]} * f + B^{[3]} \\
  \hat{y} &= \text{softmax}(Z^{[3]})
\end{align*}
\]  

In order to save space, I will not give a complete derivation of the formulas. The curious reader can find out for himself in [60] and [61]. Let \( \hat{y} \) be the prediction of the model and \( y \) be the actual labels. Let \( W^{[3]} \) be the weights and \( B^{[3]} \) be the bias for the 3-rd layer. We will then use the following equations for backpropagation:

\[
\begin{align*}
  \frac{\partial L}{\partial \hat{y}} &= (\hat{y} - y) \\
  \frac{\partial L}{\partial W^{[3]}} &= \frac{\partial L}{\partial \hat{y}} f^T \\
  \frac{\partial L}{\partial B^{[3]}} &= \frac{\partial L}{\partial \hat{y}} \\
  \frac{\partial L}{\partial f} &= (W^{[3]})^T * \frac{\partial L}{\partial \hat{y}}
\end{align*}
\]  

In the second half of the network, we need to propagate the error back through the flatten layer. As flatten operation does not change any values, we just simply reshape the gradient to the shape of \( P^{[2]} \):

\[
\frac{\partial L}{\partial P^{[2]}} = \text{reshape}(\frac{\partial L}{\partial f}, P^{[2]}.shape)
\]  

There are no weights to learn, so we just focus on propagating the gradient. During backpropagation, the gradient is only assigned to the pixel with the highest value in the pooling block:

\[
\frac{\partial L}{\partial Z^{[2]}_{mn}} = \begin{cases} \frac{\partial L}{\partial P^{[2]}_{ki}}, & \text{if } Z^{[2]}_{mn} \text{ is the max element} \\ 0, & \text{otherwise} \end{cases}
\]  

In order to calculate \( \frac{\partial L}{\partial C^{[3]}} \) for the activation layer, we will utilize the chain rule for derivatives:

\[
\text{Figure 3.8} \text{ An example of CNN architecture, consisting of two convolutional, one avg. pooling, one max pooling, and one fully connected layer. Activation function is ReLU.}
\]
\[
\frac{\partial L}{\partial C[2]} = \frac{\partial L}{\partial Z[2]} \frac{\partial Z[2]}{\partial C[2]}
\]

where \(\frac{\partial Z[2]}{\partial C[2]}\) is the derivation of the activation function, which in our case is ReLU. Its derivation can be calculated as:

\[
\frac{\partial Z[2]}{\partial C[2]} = ReLU(C[2])' = \begin{cases} 
1, & C_{mn}^{[2]} > 0 \\
0, & C_{mn}^{[2]} < 0 
\end{cases}
\]

Finally, we get to the convolutional layer. One of the nice features is that the gradient of the loss function with respect to weights itself is the result of convolution between input and the gradient from the previous layer. We can obtain the gradient of the loss function with respect to the input pixel by convolving the previous gradient with the flipped version of a kernel:

\[
\frac{\partial L}{\partial K[2]} = P[1] * \frac{\partial L}{\partial C[2]} \\
\frac{\partial L}{\partial P[1]} = \frac{\partial L}{\partial C[2]} * flip_{180}(K[2])
\]

The first half of the network will differ only in the average pooling layer. In average pooling, we divide the backpropagated gradient by the area of the pooling block and distribute it equally among all pixels within that block. Remaining calculations for the first half:

\[
\frac{\partial Z[1]}{\partial C[1]} = ReLU(C[1])' = \begin{cases} 
1, & C_{mn}^{[1]} > 0 \\
0, & C_{mn}^{[1]} < 0 
\end{cases}
\]

\[
\frac{\partial L}{\partial K[1]} = Input * \frac{\partial L}{\partial C[1]}
\]

And now, when we have all the gradients, we are finally able to update weights:

\[
K[1] = K[1] - \eta \frac{\partial L}{\partial K[1]}
\]

, where \(\eta\) is the learning rate.
Chapter 4

Datasets

This chapter describes existing image malware datasets and one binary dataset, which can be converted to image representation and used for CNN model training as well.

4.1 SOREL-20M

SOREL-20M (Sophos-ReversingLabs 20 Million) - is a malware dataset for ML purposes created by the anti-virus company Sophos in 2020. The dataset contains 20 million Windows PE executables, including 10 million disarmed malware samples.

4.1.1 GitHub repository

The project’s GitHub repository is licensed under Apache 2.0 license. It contains supportive scripts and instructions about how to use them, environment requirements, license information, terms of conditions, and answers to frequently asked questions. Among the presented scripts, there are those for training and evaluating ML models ('train.py' and 'evaluate.py'). Also, the authors provide two baseline models: a Pytorch feed-forward neural network (FFNN) model and a LightGBM gradient-boosted decision tree model. Their structure can be found in 'nets.py' and 'lightgbm_config.json' respectively.

For the purposes of this thesis, we are interested in two scripts that work with SQLite3’s "meta.db" database of malware metadata and load the dataset - 'dataset.py'. Parts of their code will later be used during the implementation phase. The database structure is described later in the 4.1.3.

4.1.2 AWS S3 bucket

The core files of the dataset are available via AWS S3 bucket: s3://sorel-20m/09-DEC-2020/. It contains such things as trained FFNN and LightGBM model (checkpoints/), performance results for each model (results/), extracted LightGBM features (lightGBM-features/), zlib compressed malware binaries (binaries/), and dataset metadata: SQLite db, Ember 2.0 and pefile features (processed-data/). The full bucket structure is shown in the figure below.
The most important for this work is the binaries/ folder, which contains around 9,919,251 disarmed malware samples ~ 8TB of data. These samples were collected by Sophos-ReversingLabs from January 1, 2017, to April 10, 2019.

### 4.1.3 Metadata structure

The SQLite database schema for the meta.db file within the 'processed-data' sub-directory is as follows:

```sql
CREATE TABLE meta (
    sha256 text primary key,
    is_malware SMALLINT,
    rl_fs_t DOUBLE,
    rl_ls_const_positives INTEGER,
    adware INTEGER,
    flooder INTEGER,
    ransomware INTEGER,
    dropper INTEGER,
    spyware INTEGER,
    packed INTEGER,
    crypto_miner INTEGER,
    file_infector INTEGER,
    installer INTEGER,
    worm INTEGER,
    downloader INTEGER
);
```
Benign samples

- sha256 – The SHA256 hash of the unaltered file (note that all provided files have been disarmed).
- is_malware – A value of 0 denotes a benign sample, while a value of 1 indicates malware.
- rl_fs_t – The initial occurrence (in Unix epoch time) of a particular sample (unique per SHA256) within the ReversingLabs feed.
- rl_fs_const_positives – The aggregate count of ‘positive’ detections (i.e., malware) from all detectors at the latest timestamp when the samples were observed. This assumes that more recent scans are of higher quality due to updated signatures, etc.
- adware, flooder, ransomware, dropper, spyware, packed, crypto_miner, file_infector, installer, worm, downloader – Each token in the detection names corresponds to a specific tag, such as adware, flooder, ransomware, dropper, spyware, packed, crypto_miner, file_infector, installer, worm, or downloader. A value greater than zero indicates the presence of the tag in the detection name. Higher values may suggest a higher degree of certainty regarding the presence of the tag.

4.2 Benign samples

For malware detection, aka malicious vs. benign classification, we will need a non-malware file source. Benign files rarely become a part of the dataset due to copyright protection. In case of visualization images are resized to fixed size, which causes interpolation and information loss, so images of benign-ware can be published as part of dataset as it wont be possible to restore the original content. The benign samples for this dataset were collected from various sources.

4.3 SORRY dataset

SOREL Research imagerY (SORRY) dataset is a dataset created as part of this master thesis. It consists of 7700 training and 3850 testing malware samples from the original SOREL-20M dataset, converted into RGB images. It also contains 7700 benign samples for training and 3850 samples for testing gathered from various sources.

<table>
<thead>
<tr>
<th>No</th>
<th>Malware family</th>
<th>Training</th>
<th>Testing</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Adware</td>
<td>700</td>
<td>350</td>
<td>1050</td>
</tr>
<tr>
<td>2</td>
<td>Flooder</td>
<td>700</td>
<td>350</td>
<td>1050</td>
</tr>
<tr>
<td>3</td>
<td>Ransomware</td>
<td>700</td>
<td>350</td>
<td>1050</td>
</tr>
<tr>
<td>4</td>
<td>Dropper</td>
<td>700</td>
<td>350</td>
<td>1050</td>
</tr>
<tr>
<td>5</td>
<td>Spyware</td>
<td>700</td>
<td>350</td>
<td>1050</td>
</tr>
<tr>
<td>6</td>
<td>Packed</td>
<td>700</td>
<td>350</td>
<td>1050</td>
</tr>
<tr>
<td>7</td>
<td>Crypto miner</td>
<td>700</td>
<td>350</td>
<td>1050</td>
</tr>
<tr>
<td>8</td>
<td>File infector</td>
<td>700</td>
<td>350</td>
<td>1050</td>
</tr>
<tr>
<td>9</td>
<td>Installer</td>
<td>700</td>
<td>350</td>
<td>1050</td>
</tr>
<tr>
<td>10</td>
<td>Worm</td>
<td>700</td>
<td>350</td>
<td>1050</td>
</tr>
<tr>
<td>11</td>
<td>Downloader</td>
<td>700</td>
<td>350</td>
<td>1050</td>
</tr>
<tr>
<td>12</td>
<td>Benign</td>
<td>7700</td>
<td>3850</td>
<td>11550</td>
</tr>
</tbody>
</table>

Table 4.1 Malware family distribution in SORRY dataset

As SOREL contains millions of samples, it is possible to distribute classes evenly, with almost any amount of samples per class. To split the dataset into training and testing parts, timestamps
recommended by the creators of SOREL-20M were used. The training dataset contains malware samples seen in the ReversingLabs feed before November 30, 2018. The testing dataset contains malware samples seen after January 12, 2019. [66]

The process of collecting this dataset involved the following steps. Firstly, the script retrieves malware metadata from meta.db within a specified time period. Then, each binary from this list is downloaded from S3, decompressed, converted into an image, and resized to the given dimensions. This image is then saved to a folder corresponding to a malware label. This structure allows to conveniently load a dataset into an ML framework, for example by using image_dataset_from_directory method in Tensorflow [67] [68].

Figure 4.2 A visualization of each class in the SORRY dataset.

Metadata does not contain a clearly defined class label. Instead, it contains something called tag vector as described in [4.3]. The final malware label is derived from the tag vector by taking the tag name of the maximal element. For example, in the picture [4.3] we can see that this malware sample has a value of 6 in the "flooder" column, a value of 1 in the "spyware" column, and a value of 3 in the "packed" column. So the final label for this sample will be derived as "flooder". If multiple classes have the same maximum value, the first class encountered will be selected.
Now, let’s take a closer look at some images from our dataset. At 4.4 you can see visualized ransomware samples. In the image representation, we can clearly see that malware executables contain PDF icons. Given the specifics, we can assume that these icons were used to imitate legitimate PDF documents and increase the chances that the victim will launch malware.

![Figure 4.3 An example of the malware sample containing multiple tags](image)

**Figure 4.3** An example of the malware sample containing multiple tags

**Figure 4.4** Two visualized ransomware samples. Images from the SORRY dataset

### 4.4 Malevis

The Malevis dataset is RGB based ground truth dataset collected by the Multimedia Information Lab of Hacettepe University Computer Engineering, cooperated with COMODO Inc. [69]

The dataset contains byte images of 26 (25+1) classes, where 25 classes are associated with different types of malware, and the remaining one class represents benign software. To create this dataset, authors used bin2png script [20], which was developed by Sultanik and is publicly available on GitHub. [69]

They took the files supplied to them by Comodo Inc. and converted them into 3-channel RGB form while fixing the image width at 224 and 300px. As most convolutional neural network architectures usually work with square images, obtained vertically long images were then resized to the square resolutions using Lanczos interpolation. [69]
Figure 4.5 A visualization of each class in the Malevis [19] dataset.

In total, the dataset consists of 9100 training and 5126 validation RGB images. Every training class has 350 image samples, whereas the number of images in validation set vary. [69] Considering that the core objective of malware detection/recognition lies in distinguishing benign samples from malware, the authors allocated a significantly larger set of "legitimate" samples in the validation set (350 vs. 1482). [69] Shows different malware families from the Malevis dataset and their visualization. We can see that each family has its own visual pattern, which is different from others.

Below is the table 4.2 containing the full list of malware families from the Malevis dataset, along with the sample counts in the training and testing splits.
<table>
<thead>
<tr>
<th>No</th>
<th>Malware family</th>
<th>Malware type</th>
<th>Training</th>
<th>Testing</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Win32/Adposhel</td>
<td>Adware</td>
<td>350</td>
<td>144</td>
<td>494</td>
</tr>
<tr>
<td>2</td>
<td>Win32/Agent-fyi</td>
<td>Trojan</td>
<td>350</td>
<td>120</td>
<td>470</td>
</tr>
<tr>
<td>3</td>
<td>Win32/Allaple.A</td>
<td>Worm</td>
<td>350</td>
<td>128</td>
<td>478</td>
</tr>
<tr>
<td>4</td>
<td>Win32/Amonetize</td>
<td>Adware</td>
<td>350</td>
<td>147</td>
<td>497</td>
</tr>
<tr>
<td>5</td>
<td>Win32/Androm</td>
<td>Backdoor</td>
<td>350</td>
<td>150</td>
<td>500</td>
</tr>
<tr>
<td>6</td>
<td>Win32/AutoRun-PU</td>
<td>Worm</td>
<td>350</td>
<td>146</td>
<td>496</td>
</tr>
<tr>
<td>7</td>
<td>Win32/BrowseFox</td>
<td>Adware</td>
<td>350</td>
<td>143</td>
<td>493</td>
</tr>
<tr>
<td>8</td>
<td>Win32/Dinwod!Rfn</td>
<td>Trojan</td>
<td>350</td>
<td>149</td>
<td>499</td>
</tr>
<tr>
<td>9</td>
<td>Win32/Elex</td>
<td>Trojan</td>
<td>350</td>
<td>150</td>
<td>500</td>
</tr>
<tr>
<td>10</td>
<td>Win32/Exprio-H</td>
<td>Virus</td>
<td>350</td>
<td>150</td>
<td>500</td>
</tr>
<tr>
<td>11</td>
<td>Win32/Fasong</td>
<td>Worm</td>
<td>350</td>
<td>150</td>
<td>500</td>
</tr>
<tr>
<td>12</td>
<td>Win32/HackKMS.A</td>
<td>Trojan</td>
<td>350</td>
<td>149</td>
<td>499</td>
</tr>
<tr>
<td>13</td>
<td>Win32/Flux!IK</td>
<td>Worm</td>
<td>350</td>
<td>150</td>
<td>500</td>
</tr>
<tr>
<td>14</td>
<td>Win32/Injector</td>
<td>Trojan</td>
<td>350</td>
<td>145</td>
<td>495</td>
</tr>
<tr>
<td>15</td>
<td>Win32/InstallCore.C</td>
<td>Adware</td>
<td>350</td>
<td>150</td>
<td>500</td>
</tr>
<tr>
<td>16</td>
<td>Win32/MultiPlug</td>
<td>Adware</td>
<td>350</td>
<td>149</td>
<td>499</td>
</tr>
<tr>
<td>17</td>
<td>Win32/Neoreklami</td>
<td>Adware</td>
<td>350</td>
<td>150</td>
<td>500</td>
</tr>
<tr>
<td>18</td>
<td>Win32/Neshta</td>
<td>Virus</td>
<td>350</td>
<td>147</td>
<td>497</td>
</tr>
<tr>
<td>19</td>
<td>Win32/Regrun.A</td>
<td>Trojan</td>
<td>350</td>
<td>135</td>
<td>485</td>
</tr>
<tr>
<td>20</td>
<td>Win32/Salify</td>
<td>Virus</td>
<td>350</td>
<td>149</td>
<td>499</td>
</tr>
<tr>
<td>21</td>
<td>Win32/Snarasite.D!tr</td>
<td>Trojan</td>
<td>350</td>
<td>150</td>
<td>500</td>
</tr>
<tr>
<td>22</td>
<td>Win32/Stantinko</td>
<td>Backdoor</td>
<td>350</td>
<td>150</td>
<td>500</td>
</tr>
<tr>
<td>23</td>
<td>VBA/Hilium.A</td>
<td>Virus</td>
<td>350</td>
<td>150</td>
<td>500</td>
</tr>
<tr>
<td>24</td>
<td>Win32/VBKrypt</td>
<td>Trojan</td>
<td>350</td>
<td>146</td>
<td>496</td>
</tr>
<tr>
<td>25</td>
<td>Win32/Vilsel</td>
<td>Trojan</td>
<td>350</td>
<td>146</td>
<td>496</td>
</tr>
<tr>
<td>26</td>
<td>Other</td>
<td>Benign</td>
<td>350</td>
<td>1482</td>
<td>1832</td>
</tr>
</tbody>
</table>

Table 4.2 Malware family distribution in Malevis dataset
Chapter 5
Implementation

5.1 Prerequisites

In this section, I will describe the methodology that will be used to train and evaluate the performance of the model.

5.1.1 Transfer learning

It is known that accuracy of the network depends on its depth. But the deeper the neural network and the more parameters it has, the longer its training takes. One common method is to use a neural network that has already been trained on a different but similar task and tweak it just a little for your purposes. This approach is called transfer learning. In this case, the fully connected layer is usually removed and replaced with a new one, which is retrained on the desired dataset. In this case, the upper layers of the neural network remain untouched and act as a feature extractor. If, after training the associated layer, the achieved accuracy is not sufficient, it is possible to unfreeze the upper layers and train weights in hidden layers. To avoid overfitting, a lower learning rate is usually selected. This process is called fine-tuning.

5.1.2 Classifier selection

For a model selection I applied transfer learning technique and initialized MobileNetV2 architecture with weights trained on the ImageNet dataset. To evaluate the effectiveness of my solution, I will compare performance metrics of CNN with Random Forest Classifier. Feature vectors were extracted from executables by using features.py script from the EMBER dataset.

5.2 Performance metrics

For each model and each dataset I’ve measured the next set of metrics: accuracy, precision, recall and F1-score. These metrics are widely used by ML researchers.

- True Positive (TP): instances where the model correctly predicts class A and the true label is also class A.
- True Negative (TN): instances that are correctly classified as not belonging to class A (true negatives for class A).
False Positive (FP): instances that are incorrectly classified as belonging to class A when they actually belong to other classes.

False Negative (FN): instances that are incorrectly classified as not belonging to class A when they actually belong to class A.

Accuracy is calculated as the proportion of correctly predicted outcomes to the total number of predictions. Measuring accuracy is crucial as it provides an overall assessment of the model’s performance in making correct predictions across all classes or categories. \[ \text{Accuracy} = \frac{\text{True Positives} + \text{True Negatives}}{\text{Total Number of Samples}} \times 100\% \] (5.1)

Precision measures the fraction of positive predictions that are accurate, calculated by dividing the number of true positives by the total number of positive predictions. High precision indicates that the model minimizes false positives, which is essential for ensuring the reliability of the model’s predictions. \[ \text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \] (5.2)

Recall, also known as sensitivity or true positive rate, evaluates the model’s capability to accurately detect all instances of a specific class. It’s computed by dividing the number of true positives by the total number of actual positives. High recall indicates that the model effectively identifies a significant proportion of positive instances, minimizing the risk of false negatives. \[ \text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \] (5.3)

The F1-score, also known as the balanced F-score or F-measure, represents the harmonic mean of precision and recall. It is computed as the weighted average of precision and recall, providing a single metric that considers both aspects of a model’s performance. By combining precision and recall into a single metric, the F1-score provides a balanced assessment of the model’s ability to make accurate positive predictions while minimizing false positives and false negatives. \[ F_1 = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \] (5.4)

5.3 Training and testing

5.3.1 MaleVis dataset

I started training the convolutional neural network by loading the model from the library. For experiments, I used the machine learning framework Tensorflow. I used a dropout layer with a probability of 0.2 percent, global average pooling and the Adam optimizer with a learning rate of 0.001. As we can see in the graph 5.1, the accuracy immediately began to increase rapidly. But as we progressed, this speed slowed down. After 10 epochs, I unfrozen the top layers and started fine tuning. As can be seen in the graph, after the green bar the accuracy increased sharply and exceeded 90 percent. After this, the accuracy increased further to 96.2 percent. Also on the graph we can see that the value of the price function decreased with each iteration, which is a good sign.
Next we can see the confusion matrix. The classes that are predicted correctly are on the diagonal, but you can notice that there are slight deviations. Next come the confusion matrices for training, validation 5.2 and testing 5.3.

**Figure 5.1** Model accuracy graph, which clearly shows the moment fine tuning begins

**Figure 5.2** Confusion matrices for training and validation splits
And here we see the neural network predictions and the correct labels. As we can see from this picture, all labels from this batch were predicted correctly.
Conclusions

5.4 Conclusions

Convolutional neural networks are a very powerful tool in image recognition, and the use of transfer learning can help save time or avoid retraining in the case of a small dataset. The goal of this work was not to create the best neural network architecture, but rather to try to look at it all from the inside. Find out how the system works from the inside and how it learns. Some knowledge was difficult to find even in scientific articles and books.

Overall, the model performed very well. Of course, it did not achieve the accuracy of the state of the art solution, but it provided a compromise between the expended computing power and accuracy. As a result, the neural network was able to achieve an accuracy of 96.2 percent, precision 95.0, recall 0.93 and F1-score of 0.94.

I also managed to learn how to visualize binary files. In this form, new ways to work with them open up. For example, the use of various transformations or computer vision methods, such as object recognition or GAN. This may become part of subsequent scientific work.

<table>
<thead>
<tr>
<th>Model</th>
<th>Accuracy</th>
<th>Precision</th>
<th>Recall</th>
<th>F1-score</th>
</tr>
</thead>
<tbody>
<tr>
<td>MobileNetV2</td>
<td>0.962</td>
<td>0.95048</td>
<td>0.934343</td>
<td>0.942342</td>
</tr>
<tr>
<td>Resnet101 [19]</td>
<td>no data</td>
<td>no data</td>
<td>no data</td>
<td>no data</td>
</tr>
<tr>
<td>Densenet121 [19]</td>
<td>0.962</td>
<td>no data</td>
<td>no data</td>
<td>no data</td>
</tr>
<tr>
<td>Densenet169 [19]</td>
<td>0.962</td>
<td>no data</td>
<td>no data</td>
<td>no data</td>
</tr>
<tr>
<td>Densenet201 [19]</td>
<td>0.962</td>
<td>no data</td>
<td>no data</td>
<td>no data</td>
</tr>
</tbody>
</table>

Table 5.1 Performance metrics comparison


6. BONFANTE, Guillaume; KACZMAREK, Matthieu; MARION, Jean-Yves. Control Flow Graphs as Malware Signatures. 2007.


9. NAROUUEI, Masoud; AHMADI, Mansour; GIACINTO, Giorgio; TAKABI, Daniel; SAMI, Ashkan. DLLMiner: Structural Mining for Malware Detection. Security and Communication Networks. 2015, vol. 8, n/a–n/a. Available from DOI: 10.1002/sec.1255


18. JAIN, Mugdha; ANDREOPoulos, William; STAMP, Mark. CNN vs ELM for Image-Based Malware Classification. 2021.


20. Bin2Png: GitHub repository. [N.d.]. Available also from: https://github.com/ESultanik/bin2png


29. Pe file: Github repository. [N.d.] Available also from: https://github.com/erocarrera/pefile


34. SARVAM. [N.d.] Available also from: http://sarvam.ece.ucsb.edu


45. YE, Yanfang; LI, Tao; CHEN, Yong; JIANG, Qingshan. Automatic malware categorization using cluster ensemble. In: 2010, pp. 95–104. Available from DOI: 10.1145/1835804.1835820


61. PATEL, Jay. *Backpropagation for Softmax and Multi-Class Classification — Complete Mathematical Derivation*. Youtube. Available also from: [https://www.youtube.com/watch?v=f-nW8cSa_Ec](https://www.youtube.com/watch?v=f-nW8cSa_Ec).


64. PATEL, Jay. *Backpropagation in CNN - Part 1*. Youtube. Available also from: [https://www.youtube.com/watch?v=Pn7RK7tofPg](https://www.youtube.com/watch?v=Pn7RK7tofPg).


67. MARTÍN ABADI; ASHISH AGARWAL; PAUL BARHAM; EUGENE BREVDO; ZHIFENG CHEN; CRAIG CITRO; GREG S. CORRADO; ANDY DAVIS; JEFFREY DEAN; MATTHIEU DEVIN; SANJAY GHEMAWAT; IAN GOODFELLOW; ANDREW HARP; GEOFFREY IRVING; MICHAEL ISARD; JIA, Yangqing; RAFAŁ JOZEFOWICZ; ŁUKASZ KAISER; MANJUNATH KUDLUR; JOSH LEVENBERG; DANDELION MANÉ; RAJAT MONGA; SHERRY MOORE; DEREK MURRAY; CHRIS OLAH; MIKE SCHUSTER; JONATHON SHELNS; BENÖIT STEINER; ILYA SUTKESKEV; KUNAL TALWAR; PAUL TUCKER; VINCENT VANHOUCKE; VIJAY VASUDEVAN; FERNANDA VIÉGAS; ORIOL VINYALS; PETE WARDEN; MARTIN WATTENBERG; MARTIN WICKE; YUAN YU; XIAOQIANG ZHENG. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015. Available also from: [https://www.tensorflow.org/](https://www.tensorflow.org/). Software available from tensorflow.org.

68. *TensorFlow v2.16.1. tf.keras.preprocess.imaimage_dataset_from_directory*. [N.d.]. Available also from: [https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image_dataset_from_directory](https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image_dataset_from_directory).
