MA513 PROJECT

From images to malwares

BLIDI Nayel – DESCHAMPS Léo – POISSONNET Clément

Table of Contents

Introduction 2

State of the art 2

Article 1 2

Article 2 3

Neural network implementation 3

Preamble 3

Architecture 3

Libraries and data loading 3

Data preprocessing 4

Model definition 4

Model training 4

Model testing 5

Results 5

Conclusion 5

Introduction

Malware is a portmanteau word resulting from the contractions of two self-explanatory terms, “malicious” and “software”. Such programs infest personal computers in order to carry out illegal operations, undetected, to the detriment of the rightful owner of the machine.

Because these programs rely on discretion to operate successfully, they have to be structured in a way that make them unnoticeable to human (in the case of reverse-engineering) and machine (such as antivirus programs) analysis. The technique used is called “obfuscations”.

Obfuscations represents the syntactic code transformations applied to a script, which results in a different program, but that retains its functionality while being harder to understand and analyse. Coupled with code optimization algorithms, it results in a new file that is beyond understanding both for the human and the threat detection algorithm.

Consequently, to overcome this high-level jamming, there is a need to develop techniques that operate directly on the raw binary, which can’t be deeply perturbated by such obfuscation methods, and which should retain some intrinsic patterns that could be used as clues in the detection of malwares.

State of the art

This work is heavily relying on two recently published research articles:

Article 1

Towards Building an Intelligent Anti-Malware System: A Deep Learning Approach using Support Vector Machine (SVM) for Malware Classification, by Abien Fred M. Agarap, 2019

This first source presents a simple and concise description of the methodology of the concept, highlighting how raw binary malwares data converted to grey levels images, from the Malimg dataset created by Nataraj et al. in 2011, can be fed into machine learning algorithm to detect malicious programs from 25 different malwares families.

This work features results from three different methods, all relying on a SVM classifier coupled to a One-vs-All discriminator, to upgrade it from a binary classifier to a multi labels classifier. This SVM layer is used as an output layer of the three following deep networks:

- Gated Recurrent Unit: a variant of a Recurrent Neural Network architecture
- Multilayer Perceptron: a fully connected architecture
- Convolutional Neural Network: a two convolutional layers network, followed by LeakyRELU activation and maxpooling operations, which conserve the input features size, to then send it into two linear sorting layers.

The overall accuracy obtained is heavily questionable, as training accuracy show an overfitting tendency for all the models, and a best accuracy of 85% on testing dataset for the GRU model, with the CNN one being last of the podium, with a score of 77%.

Because of how surprisingly bad the results were, we decided to deepen the current state of the art study, and to look into more recent and supported research articles, as detailed below.

Article 2

Data augmentation and transfer learning to classify malware images in a deep learning context, by Niccolò Marastoni, Roberto Giacobazzi and Mila Dalla Preda, 2021

This second article is a longer one, that on top of presenting a more thorough study of the deep learning solution to the malware detection and classification problem, also introduces a strong presentation of the background of obfuscation techniques, and how the training datasets were created and somewhat sometimes flawed.

Because the purpose of the current report isn't the data itself, we won't dive into these parts, although they are actually the biggest challenge of the malware detection, rather than the classification deep learning algorithms. On top of the Malimg dataset, the paper also features the Microsoft malware dataset (MsM2015), a gigantic sample of raw malwares data from 9 different families. All these datasets are then augmented to 64x64 or 64x256 images. Note than in our work, we will be using the regular 32x32 images from the Malimg dataset.

The paper compares two deep network architecture, a CNN whose architecture is similar to the one of the Article 1 above, and a bidirectional LSTM counterpart composed of 141 and 94 units. The best results of these networks on the Malimg dataset for 64x64 inputs are 98.1% and 98.5% accuracy respectively.

Neural network implementation

Preamble

The following work aims at reproducing results somewhere between those of the first and second articles. To keep a base of comparison, and although the LSTM seems to be the architecture to go with, we'll be implementing a CNN model of similar dimensions.

The whole solution can be found in the *main.py* file joined to this report, but a few points should be noted, in order to run it:

1) Execution: file's sections are called using console inputs (sys.argv). If the file isn't run from a command prompt, then these conditions should be removed or activated manually. They are:
    a. training: runs the model training with the current parameters, and saves it (overwrites pretrained models)
    b. testing: tests the current/loaded model on the training and testing datasets
    c. pretrained: loads automatically the best saved model
2) Environment: the development environment is base conda, with a few additional libraries:
    a. sklearn
    b. tqdm
    c. torch & torchvision

Some examples of code compilation would be:

```
> python .\main.py training testing
```

```
> python .\main.py testing pretrained
```

Architecture

Libraries and data loading

The dataset is saved as a numpy zipped array (.npz). This archive is automatically loaded using the user's local path and should thus be saved in the same folder as the *main.py* file. The features and labels are stored separately into two arrays, and the labels vector is transformed into a one-hot encoded targets array, to support SoftMax classification.

The data array is un-squeezed of one dimension to fit the torch image processing conditions, that take inputs of shapes (batch_size, in_channels, height, width). In tensorflow, the inputs would have been either (batch_size, height, width) or (batch_size, height, width, in_channels).

Data preprocessing

The dataset is split into training (75%) and testing (25%) arrays, and the data is normalized using the Z-score formula, that might be not judicious when applied to images, but because these are fake pictures, then the impact is negligeable, and the trick does work (we did test pixels normalization, and the results were identical).

The datapoints are casted into torch.Tensor objects and regrouped as torch.TensorDatasets objects to process the normalization and then standardize the inputs as torch.Dataloader items.

Model definition

The Convoluted Neural Network (CNN) architecture is defined as follows:

Layer	Parameters	Number of features

Convolution	In_channels = 1
	Out_channels = 16
	Kernel_size = 3x3
	Padding = 1
	Stride = 1
		Input = 1x32x32 = 1024
		Output = 16x32x32 = 16384

Activation	ReLU()	Input=Output

Maxpooling	Kernel_size = 2x2
	Stride = 2
		Input = 16x32x32 = 16384
		Output = 16x16x16 = 4096

Convolution	In_channels = 16
	Out_channels = 64
	Kernel_size = 3x3
	Padding = 1
	Stride = 1
		Input = 16x16x16 = 4096
		Output = 64x16x16 = 16384

Activation	ReLU()	Input=Output

Flattening	num_flat_features(x)	Input = 64x16x16 = 16384
		Output = 1x16384

Linear	In_features = 16384
	Out_features = 1024
		Input = 1x16384
		Output = 1x1024

Activation	ReLU()	Input=Output

Linear	In_features = 1024
	Out_features = 25
		Input = 1x1024
		Output = 1x25

The convolution layers are set with parameters padding and stride equal to 1, so there is no data loss (stride=1), and the array retains the same dimension (padding=1 with a 3x3 kernel). Thanks to these characteristics, the training will not show any noticeable overfitting, and we do not have to add any dropout layer, as this seemed to be drastically harmful to the model presented in the Article 1.

Model training

The training follows a common torch loop, with the following parameters:

1) Loss (criterion): CrossEntropyLoss (a.k.a. categorical crossentropy loss with softmax activation / with logits)

2) Optimizer: Adam optimizer, learning_rate = 0.001
3) Number of epochs: num_epochs = [10, 100, 1000]

The model's weights and loss history are then saved in the /Models/ folder as .pth and .png files respectively.

Model testing

The default testing (when run after training a model) executes two independent loops, which evaluate both the training and testing datasets, and return the accuracy of the network, i.e. the percentage of good classification predictions.

When no model was trained during the script execution:

1) The user may choose which model to load by choosing the number of epochs the model was trained over,
2) If "pretrained" was one of the console's argv, then one pretrained model with the largest number of epochs is loaded.

In these two events, the training and testing accuracy are equivalent because the data was reshuffled at the beginning if the script.

Results

By running the previous testing part, the obtained results are:

Number of epochs	**Training Accuracy**	**Testing Accuracy**
10	40.1%	40.3%
100	**96.0%**	**96.3%**
1000	95.7%	95.6%

Unsurprisingly, our results are way above those obtained in Article 1, but we didn't quite reach the peak accuracy demonstrated in Article 2, as we remain roughly 2% below their results.

One obvious explanation of this phenomena would be the dimension of the datapoints, as our inputs were only of size 32x32, whereas those from Malimg were augmented to 64x64 and 64x256, thus giving a wider feature extraction potential.

Another point to consider would be the complexity of the networks, as the model's convolution layers used by Marastoni & al. have 32 and 64 filters respectively. A deeper network may extract more features, which may be crucial in the classification of underrepresented family members, as highlighted in the second article.

Conclusion

This study successfully presented a solution in the detection of malware from its raw binary data. While the implemented CNN demonstrates promising results, future work could involve experimenting with larger image sizes, more complex network architectures, and incorporating techniques such as transfer learning, to then compare it to other state of the art techniques.

In essence, the report contributes to the ongoing effort to develop intelligent anti-malware systems, highlighting the challenges posed by evolving obfuscation methods. This work further emphasizes the need for interdisciplinary approaches, combining insights from cybersecurity, machine learning, and data augmentation techniques to stay ahead in the ongoing battle against malicious software.