A Study on Computational Performance of Malware Detection Using Convolutional Neural Networks

Pedro Leandro La Rotta
Department of Mechanical Engineering
Columbia University, New York, NY

Federico Tondolo
Department of Computer Science
Columbia University, New York, NY

***Abstract*—Employing Convolutional Neural Networks (CNN) for malware detection is an active area of research mostly focusing on applying transfer learning to building highly accurate models. These models take in an image representation of the bytecode or the compiled executable file and are able to classify the type of attack present. While near-perfect accuracy has been proven by several groups, no studies to our knowledge have focused on the computational performance of these models. In this work we will evaluate the computational and memory loads of the MalConv CNN architecture, and we employ several high-performance computing strategies to improve inference time.**

***Index Terms*—HPC, Cybersecurity, deep learning**

I. Introduction

It is estimated that 1 in 2 Americans had their accounts breached in 2021. Cyberattacks remain a major security risk today as nearly 1 in 10 US organizations do not have insurance against cyberattacks [cite]. Malware is one of the primary vehicles for cyberattacks, and it refers to software that is intentionally designed to harm, disrupt or gain unauthorized access to a computer system. Within cybersecurity, malware detection is an important area of research given both the global rise in online presence and the general rise in complexity of cyberattacks.

A. Malware Detection

Malware detection traditionally has been classified as either signature-based or behavior-based, where the former looks for patterns in the digital signature of the program that may be correlated to malware, and the latter seeks to identify malicious patterns in the actual program behavior. Because the input space for this problem is so large (many different file types, file lengths, etc.), traditional classification is challenging for this problem. Designing the appropriate feature encodings that generalize to diverse attacks is challenging and can be easily undermined by more sophisticated attacks.

B. Deep learning for Malware Detection

The difficulty in designing feature encodings for malware detection makes deep learning a powerful strategy since less bias would come into the model from the designer's decisions. Deep learning models have shown remarkable aptitude for representing high dimensional data in lower dimensional space through convolution filters.

Efforts to use deep learning for malware detection have been largely successful with some groups reporting near perfect accuracy and impressive classification metrics. One approach embeds executable files in grayscale images and feeds them through an image classification network. This approach demonstrated the incredible power of these models (98% test accuracy), but their training and results all came from a large and preprocessed dataset of binary file images. The main challenge in developing deep learning models for this task is simply generating (and handling) a large and diverse enough dataset. MalConv is the model architecture used in this study, and it consists of an embedding layer followed by 2 parallel convolution layers feeding into a sigmoid activation. The output of this model is a 1 or 0 label, and its original study reported an accuracy of 80% on a large, diverse custom dataset.

C. Datasets

EMBER is one of the best datasets for training malware classifiers. It consists of 20 million raw executable files, and it holds nearly 8TB of .exe files and metadata. Unfortunately, the size of this archive and its age (inception in 2017) made it an unappealing choice for this project. Other datasets do exist, but most of the other datasets hold neutralized or modified viruses, and are often filled with old, well characterized attacks.

II. Models and Dataset Description

A. Model

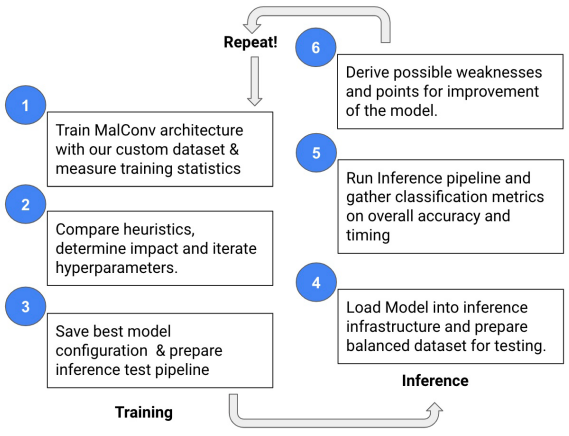
With regards to the model we used in our exploration, we settled on the MalConv model, a moderately simple and portable model with circa 1,042, 953 trainable model parameters constructed through the combination of an embedding layer, 2 convolutional layers, one of which is fed through a sigmoid function, which are then combined, pooled, fully connected and sent through an activation function ahead of providing an output (we used both ReLU and Sigmoid activation functions at this final step to examine the effect these variations might have on model performance). In the course of hyperparameter sweeping for the model we changed a number of aspects of this rather broad network description (while keeping the general structure identical for the sake of comparability) including learning rate, batch size, window size, and representation size. The first two of these are rather self explanatory, as for the latter half, the window size reflected the size of the segmented chunks of the executable which the model iterated through during training for each sample,

in essence how many bytes at a time it would be observing; while the representation size signified the dimension of the executable representation at the pooling layer of the model, a larger dimension resulting in substantially slower computation time during inference but proving an ampler representational medium with which to classify a sample.

B. Dataset

As for the datasets we employed, we thought it best to create our own dataset for this project as it provided us not only the greatest understanding of the underlying heuristics which our model would be training on, but also to minimize what we perceived as confounding factors when it comes to malware detection which many other of the pre-crafted datasets we found possessed. More specifically, we wanted to make sure that we not only found "whole" viruses, not truncated or similarly mutilated un-runnable executables which might clue our model in to their nature due to their de-weaponized state, but that our control, non-malware group, would similarly be, while not malicious, similar enough in general structure as to allow our model to catch upon the truly malicious properties of our target executables as opposed to some frivolous difference in software construction. We went about achieving this by focusing on freeware as our control group, now not only is this easily attainable online (as there is certainly no shortage of freeware out in the wild!) but the way freeware attacks filesystems, adding login items, persistent background processes, and interacting with OS settings seemed to us a prime adversarial dataset to ensure that our models truly identified the best metrics for authentic malware detection. Now, as for the malware itself, this was not nearly as easy to find as untouched malware is quite hard to come by given their nefarious nature. We were able to scrape a number of sources but far and away the most comprehensive we could find was DasMalwerk which provided us with very new (¡1 week old) malware to test which was also documented with metadata pertaining to its recognisability by antivirus software on the market.

III. Training Methodology



Our training and associated profiling methodology starts at the training stage where we perform an industry-standard train/test split of our dataset and run the base model through a series of epochs of training. During each of these epochs we measure not only the overall epoch training time but also the time it takes for every batch to be read into the network, the component of said time which is spent enumerating the current batch's dataloader, and the validation time for the selfsame epoch. Apart from temporal measurements, we measure batch loss for the model and batch accuracy both unto itself and as a cumulative counter for the epoch as averaged out throughout past batches.

IV. Profiling Methodology

Once training is complete we then observe the data for glaring heuristics, whether validation accuracy is similar to training, maintaining proper generalization, or whether overfitting or overparameterization might perhaps be occurring, whether dataloader enumeration is causing significant slowdowns in batch training times or whether loss is maybe plateauing due to too small a learning rate or batch size. Indeed, running the trained models through inference profiling which you will read about shortly also allowed for the identification of these blind spots. With these inferences, hyperparameters for the model are tuned from the pooling size to the learning rate to the window size to the batch size. These are modulated according to the profiling done at the previous stage to attempt to maximize any improvements observed and circumvent shortcomings.

Once the best hyperparameters for the next training model have been chosen the cycle begins anew, attempting to perfect hyperparameter initialisation. Once a satisfying combination has been found, extended training over many more epochs is run to attempt to maximize accuracy.

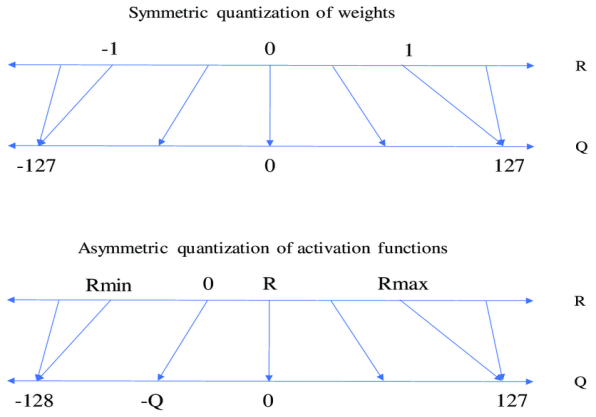
V. Performance Tuning Methodology

Inference through a neural network needs to be fast in the case of malware detection since its diagnosis should activate a series of preventative system processes to prevent an attack. For this reason, a substantial part of this study focused on implementing simple performance optimizations in pytorch.

A. Post-training Quantization

First, the weights of the trained model were trained and quantized via symmetric quantization (-127 to -128). This method seeks to reduce the memory access load of inference by storing the weights in a

lower representation like INT8 instead of full FP32.

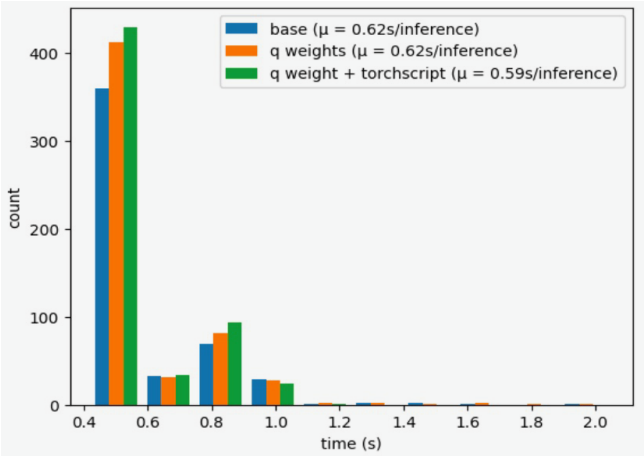


B. TorchScript

Secondly, we leveraged the TorchScript module in Pytorch to create a serialized version of our model. Scripting can give slight performance improvements with minimal effort and changes to the source code.

VI. Experimental Results

Here we can observe our inference profiling results, the behavior of our model with regards to malware classification, and the temporal improvements resultant from quantization. We note that our model maximized its accuracy and minimized its loss through the use of overly incautious classification, since most of our datapoints were not malware, classifying the vast majority as such was conducive to higher accuracy. False positives were not an issue we observed yet false negatives most definitely made an appearance in our result. Nevertheless, we can see that quantization, both of the weights alone and in conjunction with torchscript allowed for a slight improvement in model inference speed.



actual label

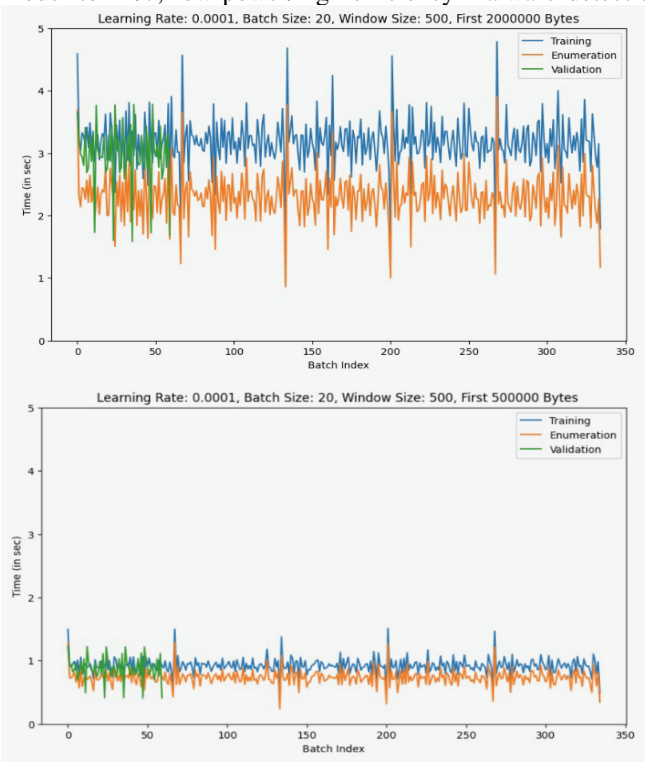
	Malware	Freeware
Malware	1	0
Freeware	150	345

predicted label

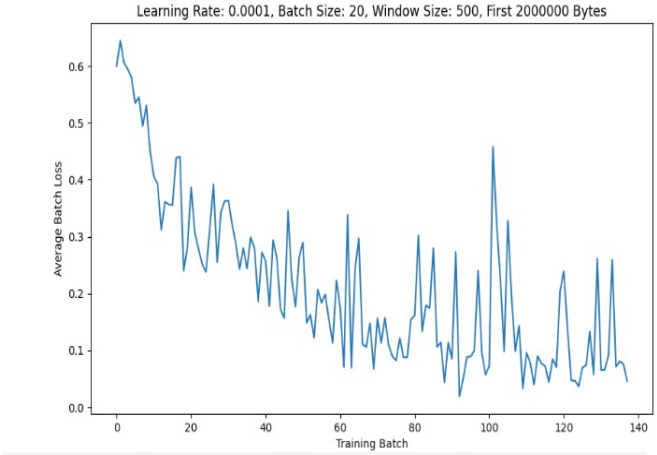
	Malware	Freeware
Malware	0	0
Freeware	151	409

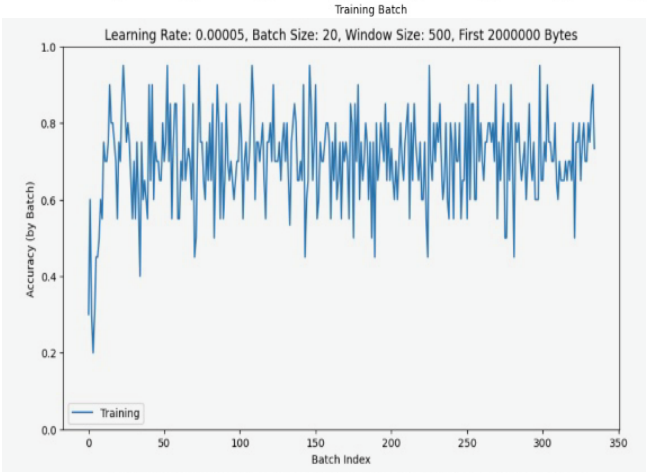
	Malware	Freeware
Malware	0	0
Freeware	151	430

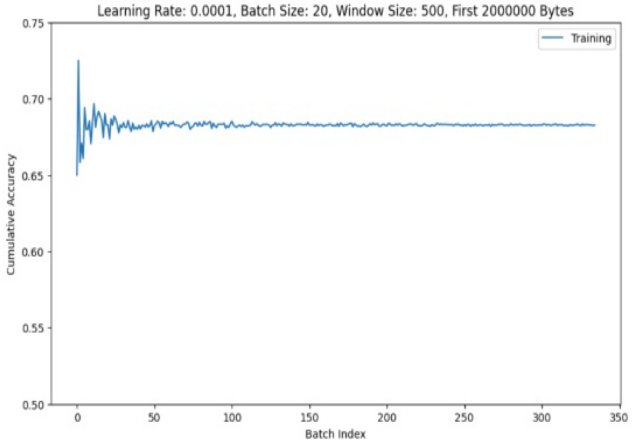
Turning our attention to the effects of our hyperparameter tuning, we can see the effect that decreasing the pooling size had on the dataloader enumeration, substantially bringing down what was the most time consuming aspect of training with what we observed to be only a small decrease in model accuracy this would be the first place to modulate to tailor a model to 24/7, low-power/high-efficiency malware detection.



Here we can see the accuracy of one our models through training, both cumulative and on a per-batch basis, each batch naturally being randomized and consequently oscillating. The accuracy of course appears to plateau in the cumulative graph as the near-constant classification ‘freeware’, whether correct or not, resulted in a very stable and predictable accuracy metric.







Finally, the loss of two different models we tested shows the gamut of possible model evolution displayed by the changing of just a few hyperparameters in an attempt to avoid vanishing gradients in this 1,000,000+ parameter model through the use of increased learning rate and/or batch size to provide a more stochastic and unstable loss projection, but a more directly converging one all the same.

VII. Conclusion

In the course of this project we observed that not only is the task of executable classification large, but the intricacies intrinsic to the scope of the task result in important considerations in model creation. Not only do we seek a model that is well suited to correctly identifying malicious executable code, but we also must contend with the risks of false positives, or, far more dangerously, false negatives. Indeed, anyone who has used an antivirus solution for any length of time has had to dismiss warnings about a completely innocent piece of code, misidentified as malware by the software solution, and while this is an inconvenience, it is a trade-off the user has chosen to make for security. However, false negatives become far more upsetting of a proposition as we have now to contend with potentially unrecognized malware free to run rampant on the user's machine. Indeed, the model we tested, irrespective of hyperparameters, suffered in part from this flaw, as while higher accuracy could be easily achieved by classifying all samples as freeware (and in so doing taking advantage of the dataset imbalance to artificially bolster its accuracy) this would result in a terrible anti-virus solution, as any user could confirm that an overly cautious antivirus solution is infinitely better than an overly lax one. This points to another obstacle in the exploration of this field however: the lack of raw training data. While malware is hard to come by, this digital "drought" renders it increasingly difficult to monitor the evolving attack vectors of malware, and it leaves antivirus solutions in a constant state of catch-up. We were able to demonstrate that convolutional neural networks have the potential for use in the field of malware recognition, training times can be reduced with proper hyperparameter selection and model quantization, and taking advantage of the long training periods of neural network one could quite easily implement a perpetual anti-virus scanner, custom-tailoring its perceptivity based on the specific digital habits of each of its users. Further research is needed to better understand ways of balancing the apparent lack of malware in most of a user's daily digital interactions to ensure faithful detection. Possibly, advancements could be achieved through the use of weighted loss calculations when a malware is correctly identified or alternatively, the combination of further heuristics such as overall executable topology in a hybrid model could provide higher accuracy, at the cost of longer training times. Nevertheless, we have sought to establish that a way to create an ever-adapting antivirus solution exists, and with the right considerations has the potential to become a radically faster and more accurate application than previous generations of malware detecting-software.

Acknowledgment

Thank you to the HPML staff and instructors for an insightful semester!

References

[1] S. Bose, T. Barao and X. Liu, "Explaining AI for Malware Detection: Analysis of Mechanisms of MalConv," 2020 International Joint Conference on Neural Networks (IJCNN), Glasgow, UK, 2020, pp. 1-8, doi: 10.1109/IJCNN48605.2020.9207322.

[2] H. Anderson and P. Roth, "EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models", in ArXiv e-prints. Apr. 2018.

[3] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro and C. K. Nicholas, "Malware detection by eating a whole exe", Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence, 2018