Malware Binary Image Classification Using Convolutional Neural Networks

John Kiger, Shen-Shyang Ho and Vahid Heydari
Rowan University, Glassboro, NJ, USA
kigerj19@students.rowan.edu
hos@rowan.edu
heydari@rowan.edu

Abstract: The persistent shortage of cybersecurity professionals combined with enterprise networks tasked with processing more data than ever before has led many cybersecurity experts to consider automating some of the most common and time-consuming security tasks using machine learning. One of these cybersecurity tasks where machine learning may prove advantageous is malware analysis and classification. To evade traditional detection techniques, malware developers are creating more complex malware. This is achieved through more advanced methods of code obfuscation and conducting more sophisticated attacks. This can make the manual process of analyzing malware an infinitely more complex task. Furthermore, the proliferation of malicious files and new malware signatures increases year by year. As of March 2020, the total number of new malware detections worldwide amounted to 677.66 million programs. In 2020, there was a 35.4% increase in new malware variants over the previous year. This paper examines the viability of classifying malware binaries represented as fixed-size grayscale using convolutional neural networks. Several Convolutional Neural Network (CNN) architectures are evaluated on multiple performance metrics to analyze their effectiveness at solving this classification problem.

Keywords: Malware Analysis, Malware Classification, Malware Visualization, Convolutional Neural Networks, Deep Learning

1. Introduction

The shortage of professional talent in the cybersecurity workforce is an issue that continues to persist as more businesses recognize the value of cybersecurity for their organization. In their 2021 Cybersecurity Workforce Study, the cybersecurity professional organization (ISC)2 reported an estimated 2.72 million global shortage of cybersecurity professionals (ISC2, 2021). Simultaneously, malware continues to spread at an unprecedented rate, with hundreds of thousands of new signatures detected every day. In 2020, Kaspersky's detection systems discovered an average of 360,000 new malicious files every day over the past 12 months—18,000 more than the previous year (a 5.2% increase) and up from 346,000 in 2018. 60.2% of those malicious files were non-specific Trojans. In general, the percentage of Trojans detected increased by 40.5% when compared to the previous year (Kaspersky Lab, 2021). Observing these trends gives the primary motivation for seeking automated solutions for malware analysis, particularly in the areas where analysis is done manually. Traditionally, malware analysis is conducted using one, or a combination of, static and dynamic malware analysis methods. However, the emergence of fileless malware has prompted a new method of memory-based analysis. Here we will provide a high-level description of current malware analysis methods to provide a deeper understanding of the current landscape.

2. Static Malware Analysis

Static malware analysis is a signature-based approach that entails deriving information about a potentially malicious executable file without the executing file. This is done through a variety of methods that attempt to derive and enumerate any signatures that classify a given executable file as malicious. Traditional static analysis methods are basic and do not offer deeper insights into a malicious file's behavior. Furthermore, sophisticated malware easily eludes traditional static malware analysis through increasingly complex code obfuscation techniques. However, these methods are the least computationally intensive and time-consuming to conduct. Therefore, static malware analysis is often the first method utilized when investigating an executable file. Typically, static malware analysis is achieved through these methods:

- Checking the executable file's hash against a known malware database, such as Virustotal. These databases are open-source and are usually evaded by serious malware developers.
- Using the 'strings' command on the executable file in a command-line interface to find any malicious keywords contained in the file's strings. This method is not effective on encrypted versions of malware.

- Manually decompiling the file in a disassembler, such as IDA, to analyze an executable file's control flow. This method requires subject matter expertise and may become intractably complex if the malicious code is severely obfuscated.

3. Dynamic Malware Analysis

Dynamic malware analysis is a behavior-based approach that entails executing a potentially malicious file in a virtualized sandbox environment and attempting to gain insights into a file's runtime behaviour through closely monitored observation. This method is the most computationally intensive of all the analysis methods due to the need to run the executable file in a virtualized sandbox. This prevents any damage a malicious file may cause to a system. Furthermore, this method may prove to be the most time-consuming as many malware files significantly delay the execution of their malicious code to circumvent antivirus software. Finally, sophisticated malware often utilizes techniques to determine if it is running in a virtualized environment and if so, will appear benign to this method of analysis. While this method is subject to be circumvented, if successful, significant insights on a malware's behaviour can be derived from the following methods:

- Monitoring malicious network traffic activities through tools, such as Wireshark. This allows the capture and deep analysis of all traffic originating from the executable file.
- Process monitoring can also provide insights into how a potentially malicious file interacts with the host system. The virtualized environment can capture calls to the host device's file system and registry to monitor any malicious activity, such as deleting or manipulating any important system resources.

4. Memory Malware Analysis

Most malware persists through an executable file or latching itself to a critical system resource. The most sophisticated and devastating of these are Advanced Persistent Threats (APT). However, stealthier attack vectors that attempt to elude traditional dynamic analysis methods, are classified as Advanced Volatile Threats (AVT). Fileless malware is classified as an AVT since it is malicious software that exclusively operates as an artifact in a system's volatile memory or RAM. Like dynamic analysis, this method requires a virtual environment with allocated RAM to capture and dump a memory image and analyze artifacts that include:

- Process lists and associated threads
- Networking information and interfaces (TCP/UDP)
- Kernel modules, including hidden ones
- Bash and command history
- System calls
- Kernel hooks

5. Related Research

This section includes a brief review of some previous image-based automated malware classification approaches.

5.1 Deep learning at the shallow end: Malware classification for non-domain experts

Le et al. (2018) presented a deep learning-based malware classification approach that required no expert domain knowledge and is based on a pure data drive approach for complex patter and feature identification. Acknowledges the lack of domain expertise when handling digital evidence in law enforcement. The research specifically cites how Artificial Intelligence (AI) can aid digital investigators. AI can expedite the investigative process and ultimately reduce the case backlog while avoiding bias and prejudice (James and Gladyshev, 2013). The contribution of this work includes a deep learning model which achieves a 98.2% accuracy in classifying raw binary files into one of 9 classes of malware. The one-dimensional representation of a raw binary file is like the image representation of a raw binary file in the work of Nataraj et al. (2011). The sequential representation allowed Le et al. (2018) to apply a Convolutional Neural Network – Bi Long Short Term Memory architecture (CNN-BiLSTM) which helped achieve better performance than using a CNN model alone.

5.2 Malware Detection by Eating a Whole EXE

The most significant finding of this work from Raff et al. (2017) is how the batch normalization method hindered the learning process for all their proposed models, in all their approaches, on all the software platforms they experimented on. When given the context of analyzing binary executables, their hypothesis is well-reasoned.

> *We hypothesize that batch norm's ineffectiveness in our model is a product of training on binary executables. The majority of contemporary deep learning research, including batch-normalization, has*

been done in the image and signal processing domains, with natural language a close second. In all of these domains, the nature of data is relatively consistent. In contrast, our binary data present a novel multi-modal nature of the byte values that can have drastically different meanings depending on the location, ranging from ASCII text, code, structured data, or even images stored for the icon. Our hypothesis is that this multi-modal nature produces multiple modes of activation, which violates the primary assumptions of batch-normalization, causing degraded performance. (Raff et al., 2017).

6. Objective

This work aims to research and perform practical experiments to determine the effectiveness of using a deep learning approach, such as a convolutional neural network, for an automated static malware detection and classification scheme. To investigate the effectiveness of different convolutional neural network (CNN) architectures for this purpose, multiple CNNs were tested, and their performance at solving this multiclass image classification problem was compared. Training time was used as a rudimentary measure of computational complexity to determine each network's applicability to run on a device with strict resource limitations, such as an IoT device. The ability to perform these tasks on a low-powered edge device was taken under consideration, as the proliferation of IoT-based malware has been seeing a significant increase. In 2019, SonicWall Capture Labs threat researchers recorded 34.3 million IoT malware attacks. In 2020, that number rose to 56.9 million, a 66% increase (SonicWall, 2021). Traditional performance metrics used for neural networks, such as accuracy, loss, F1 scores, precision, and recall, were used to determine the strength of each network architecture.

Table 1: Malimg classes, and their respective sample sizes

#	Class Name	Malware Type	Samples
1	Adailer.C	Dialer	122
2	Agent.FYI	Backdoor	116
3	Allaple.A	Worm	2949
4	Allaple.L	Worm	1591
5	Alueron.gen!J	Worm	198
6	Autorun.K	Worm:AutoIT	106
7	C2LOP.P	Trojan	146
8	C2LOP.gen!g	Trojan	200
9	Dialplatform.B	Dialer	177
10	Dontovo.A	Trojan Downloader	162
11	Fakerean	Rogue	381
12	Instantaccess	Dialer	431
13	Lolyda.AA1	Password Stealer	213
14	Lolyda.AA2	Password Stealer	184
15	Lolyda.AA3	Password Stealer	123
16	Lolyda.AT	Password Stealer	159
17	Malex.gen!J	Trojan	136
18	Obfuscator.AD	Trojan Downloader	142
19	Rbot!gen	Backdoor	158
20	Skintrim.N	Trojan	80
21	Swizzor.gen!E	Trojan Downloader	128
22	Swizzor.gen!I	Trojan Downloader	132
23	VB.AT	Worm	408
24	Wintrim.BX	Trojan Downloader	97
25	Yuner.A	Worm	800

6.1 Dataset Description

The Malimg dataset (Nataraj et al., 2011) is used in our investigation. This dataset contains 9,339 unique malware binary files, also known as portable executable (PE) files, which have been converted to PNG images. Each image in the dataset belongs to one of 25 total classes of malware. Each CNN was tasked with identifying the unique features of each malware class during training and accurately predicting the class name of a given malware binary image during validation. Table 1 shows the 25 classes and their respective number of instances.

6.2 Malimg Class Distribution and Normalization

We observe that nearly half of the dataset is comprised of only two malware classes, Allaple.A and Allaple.L. This can prove to be a severe issue when training the neural networks, so all of the class weights of the training data were normalized using the class_weight.compute_class_weight function from SciKit Learn's utils library. While this distribution may not be optimal for a research environment, it is a more realistically distributed dataset. In

the context of malware distribution, certain classes of malware can proliferate at an exponentially higher rate than other malware.

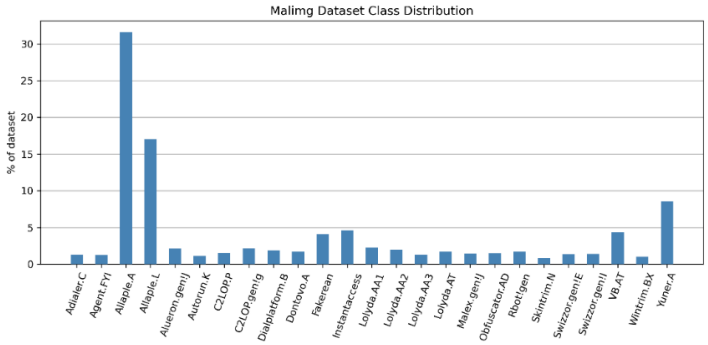

Figure 1: A bar chart demonstrating the distribution of the various classes in the Malimg dataset

6.3 Observation of a Malware Image Sample

To understand the structure of a Windows Portable Executable (PE) file, and to understand how these executable binaries were converted to images, we inspect a single, unedited sample from the Malimg dataset. A PE file consists of the following sections, which can be seen visually in Figure 1:

1. .text – The default code section which contains the executable code.
2. .rdata – The default read-only data section. String literal and C++/COM vtables are an example of items found in this section.
3. .data – The default read/write data sections. Global variables typically are stored here.
4. .rsrc – The section that contains all the resources of the module. This also includes icons that an application may use. (Microsoft, n.d.)

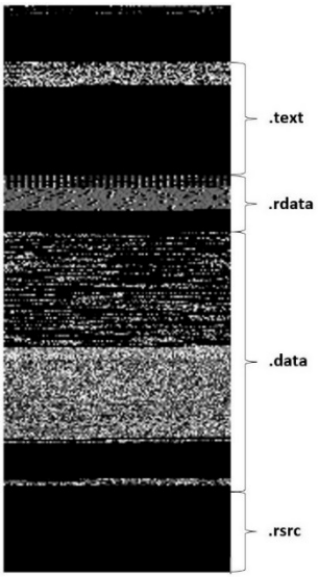

Figure 2: The code sections of a Dontovo.A Trojan

A given malware binary is read as a vector of 8-bit unsigned integers and then organized into a 2D array. This can be visualized as a grayscale image in the range [0,255] (0: black, 255: white). The width of the image is fixed, and the height is allowed to vary depending on the file size (Nataraj et al., 2011). In Figure 2, we notice multiple solid black sections, specifically in the .text section. This may represent uninitialized code; however, it is more likely to represent a common code obfuscation technique known as zero padding. The benefit of using a learning-based static analysis method is that code obfuscation techniques like this are generally ignored by each CNN.

6.4 Dataset Sample After Pre-Processing

The following visual data sample was compiled after pre-processing all images in the data set into 64x64 grayscale images. Not only does this compression retain the unique visual features that distinguish each malware class, but the image space occupied by zero padding is highly compressed and, therefore, less relevant as a significant feature of the data. Visual similarities between random samples of the same class, as well as visual similarities between similar classes are observed in Figure 3.

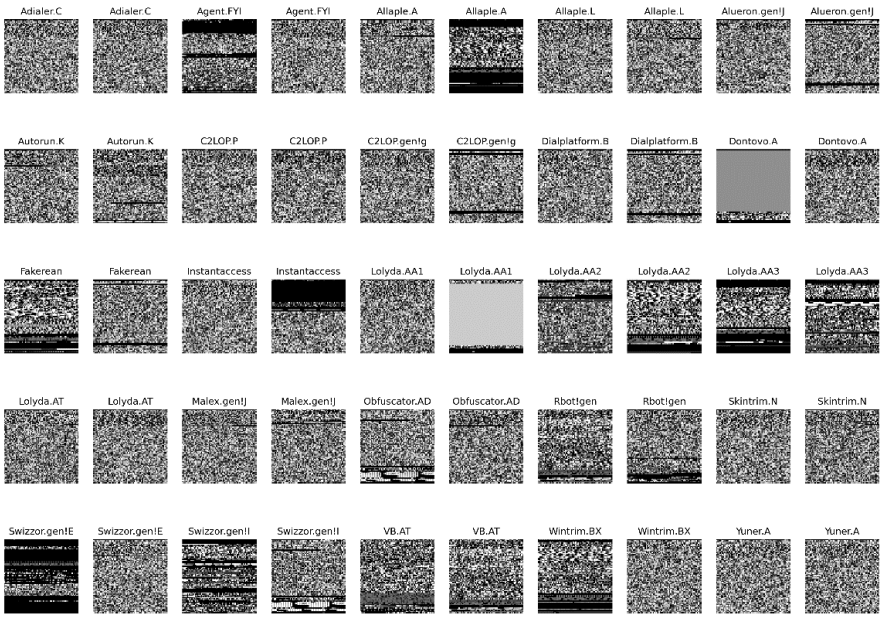

Figure 3: Samples from each class of malware from the Malimg dataset after pre-processing

7. Software and Hardware Used

7.1 Software

All experiments were done in a Python3 Juypter notebook. The machine learning libraries used were TensorFlow, Keras, and SciKit Learn. All visualizations were made using Matplotlib, and Seaborn. Other functionality, such as data processing and file I/O was accomplished using the NumPy, OS, Panda, and Collections Python libraries.

7.2 Hardware

All experiments were conducted on a 5th generation Lenovo ThinkPad X1 Carbon with an Intel Core i7-7600U with four logical cores with a 2.80Ghz clock speed and 16GB of RAM. There was no GPU acceleration utilized in these experiments.

8. CNN Performance Comparison

The performance metrics used to compare the five CNNs tested during experimentation were:

1. Total Training Time
2. Average Training time per Epoch
3. Total Accuracy
4. Total Loss
5. Weighted Average F1 Score
6. Weighted Average Precision
7. Weighted Average Recall

The purpose of our experiments was to compare the accuracy and overall strength of each network and test each network's applicability for running natively on a lower-powered device, such as an IoT or network device. One of static malware analysis strengths is the low computational resources required to analyze a potentially malicious file. Therefore, the goal of these experiments was to create a CNN that would not only classify malware accurately but with relatively great efficiency and low computational overhead. The time-based performance metrics serve as a rudimentary measure of each network's efficiency. All models used a 70/30 training/validation

split during testing. Furthermore, all models used categorical cross-entropy as the loss function and the Adam optimizer on their compilation layers. Therefore, these constants are omitted when detailing each model's architecture.

8.1 Comparison of Convolutional Model Architectures

A total of five CNN models were evaluated during our experiments. These models are split into two distinct subcategories. Models 1 and 2 use batch normalization as the normalization function, and Models 3, 4 and 5 use Local Response Normalization as the normalization function. Models 1 and 2 retain the same input size of 64x64x1, with model 2 having double the number of convolutional layers in an experiment to decrease training time between epochs. Models 3, 4 and 5 were designed to easily scale to different input sizes and designed to use a new normalization method. The latter design change was done to test the hypothesis of Raff et al. (2017) who cited the failure of batch normalization in their experiments, while the former was done to test the performance impact of using different input sizes.

8.1.1 *A Comment on Normalization Functions*

As Raff et al. (2017) cited in their work, batch normalization failed in their experiments by severely hindering the learning process on malware binaries. In their hypothesis they cited that byte values can have drastically different meanings depending on the location, ranging from ASCII text, code, structured data, or even images stored for the icon (Raff et al., 2017). This reasoning is consistent when given the context of how the batch normalization function operates and given that the malware image binaries contain structured contextual data, which was detailed in Section 4.3. In an article titled *Difference between Local Response Normalization and Batch Normalization*, Anwar, A. (2021) details the process of the batch normalization function. Before being fed to the activation function, the output of the hidden neurons is processed as by completing the following steps:

1. The entire batch B is normalized to a zero mean and unit variance
2. The mean is calculated for the entire mini-batch output.
3. The variance is calculated for the entire mini-batch output.
4. The mini-batch output is normalized by subtracting the mean and dividing by the variance.
5. Two trainable parameters are introduced, Gamma, the scale variable and Beta, the shift variable which scales and shifts the normalized mini-batch output.
6. The scaled and shifted normalized mini-batch output is fed to the activation function. (Anwar, 2021)

We hypothesize that the scaling and shifting of the batch data done during the normalization function is the reason why batch normalization may prove to hinder the learning process for contextually sensitive data, such as an image representation of a binary file. When the image is scaled and shifted during the batch normalization function, the feature's context is lost. For example, say a given sample contains a series of bits that represent a malicious API call in the .text section of a Windows PE file, if the image is scaled and shifted to where this malicious string is moved to a different section of the PE file structure, it's meaning, and context are lost to the learning algorithm.

Local response normalization does not use scaling and shifting but rather uses the concept of lateral inhibition. Lateral inhibition is a concept from neurobiology where stimulated neurons inhibit the activity of neighbouring neurons. For local response normalization, this concept of lateral inhibition is used to conduct local contrast enhancement, so the local maximum pixel values are used as the excitation for the next layers (Anwar, 2021). Local response normalization, which relies on the local context of a given pixel, rather than the shifting and scaling the image relative to the gamma and beta values in batch normalization. Local response normalization function was considered as the most appropriate normalization function over batch normalization when considering the contextual nature of the binary data. After experimenting with model 1 and 2, which used batch normalization, the remaining models utilized local response normalization. Once its effectiveness was verified in our experimentation, we then moved to experiment with different input sizes for our training and testing data. This is also the reason why models 3, 4 and 5 were designed to scale well with different sized input data.

Table 2: Comparison of the model architectures tested

Architectural Comparison					
Model No.	**Input Dimension**	**Convolutional Layers**	**Pooling**	**Normalization Function**	**Dense Layers**
1	64x64x1	30 Filters, 3x3 Kernel Size, Relu Activation 15 Filters, 3x3 Kernel Size, Relu Activation	Max Pooling 2x2 Pool Size	Batch Normalization	128 Units – Relu 50 Units – Relu 25 Units – Softmax
2	64x64x1	32 Filters, 3x3 Kernel Size, Relu Activation 32 Filters, 3x3 Kernel Size, Relu Activation 64 Filters, 3x3 Kernel Size, Relu Activation 128 Filters, 3x3 Kernel Size, Relu Activation	Max Pooling 2x2 Pool Size	Batch Normalization	256 Units – Relu 128 Units – Relu 25 Units – Softmax
3	128x128x1	50 Filters, 5x5 Kernel Size, Relu Activation 70 Filters, 3x3 Kernel Size, Relu Activation 70 Filters, 3x3 Kernel Size, Relu Activation	Max Pooling 2x2 Pool Size 1 Stride	Local Response Normalization	256 Units – Relu 25 Units – Softmax
4	64x64x1	25 Filters, 5x5 Kernel Size, Relu Activation 35 Filters, 3x3 Kernel Size, Relu Activation 35 Filters, 3x3 Kernel Size, Relu Activation	Max Pooling 2x2 Pool Size 1 Stride	Local Response Normalization	128 Units – Relu 25 Units – Softmax
5	32x32x1	15 Filters, 5x5 Kernel Size, Relu Activation 25 Filters, 3x3 Kernel Size, Relu Activation 25 Filters, 3x3 Kernel Size, Relu Activation	Max Pooling 2x2 Pool Size 1 Stride	Local Response Normalization	64 Units – Relu 25 Units – Softmax

8.2 Performance Results

Table 3: Performance results of models tested

Performance Comparison *(best results are bolded)*								
Model No.	**Total Epochs**	**Total Training Time**	**Average Training Time per Epoch**	**Total Accuracy**	**Total Loss**	**Average Weighted F1 Score**	**Average Weighted Precision**	**Average Weighted Recall**
1	25	9.23 minutes	22.16 seconds	85.22%	31.41%	0.84	0.85	0.85
2	25	**5.87 minutes**	**14.08 seconds**	88.69%	**16.17%**	0.88	0.88	0.89
3	10	3.59 hours	21.52 minutes	**95.07%**	29.98%	**0.94**	**0.94**	**0.95**
4	10	17.38 minutes	1.74 minutes	93.97%	23.12%	0.93	0.93	0.94
5	50	11.82 minutes	14.18 seconds	77.59%	135.82%	0.77	0.77	0.78

8.2.1 *Model 1 and 2 Performance Comparison*

Models 1 and 2 use very similar to architectures, both use a 64x64x1 input size, both trained for 25 epochs, and both use batch normalization as their normalization function. Due to their similarity, their performance results will be compared directly. Outside of model 5, which used a significantly reduced image input size, model 1 was the worst performing model regarding accuracy. While training only took under 10 minutes, the accuracy gave

room for improvement. Model 2 was designed to expand on model 1. First, it was designed to increase accuracy and reduce training time. While model 2 achieved both goals, with a 3.47% increase in accuracy with a 3.36-minute reduction in training time, there was concerns about its strength as a model.

In Figure 4, there was a significant deviation between model 2's validation and training accuracy starting at the 20th epoch. This is not ideal as the goal is to have the validation curve follow the training curve as closely as possible. Upon further investigation, we theorized that this could have been caused by using batch normalization or the architecture of models 1 and 2.

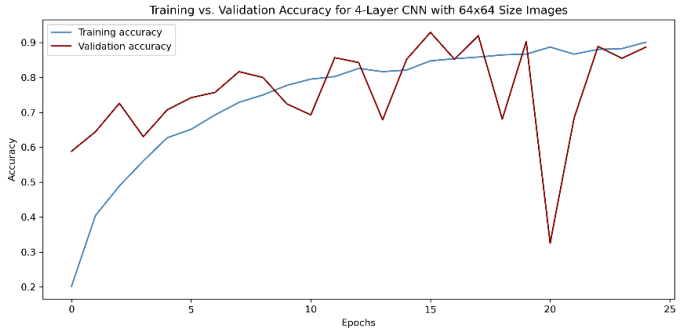

Figure 4: Training vs. validation accuracy for model 2

8.2.2 *Model 3, 4 and 5 Performance Comparison*

Models 1 and 2 exhibited significant deviation between training and validation on accuracy and loss curves. This behaviour prompted a significant restructuring of the subsequent networks, prioritizing scalability to easily test different input sizes and the switch from batch normalization to local response normalization. The first model to test this new architecture was model 3, which used a 128x128x1 input size. Doubling the input shape over the previous two model's 64x64x1 input shape increased the accuracy, with the best-recorded accuracy for model 3 being 95.07%. However, the training time was significantly longer, at nearly four hours.

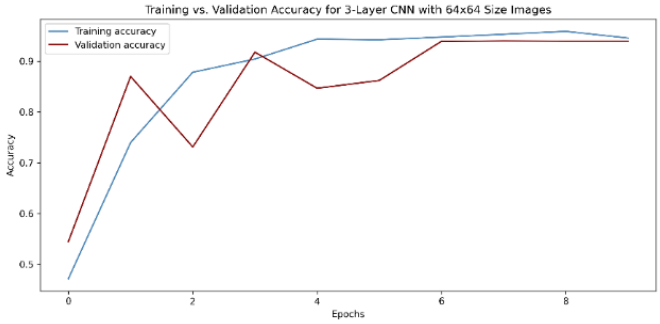

Figure 5: Training vs. validation accuracy for model 4

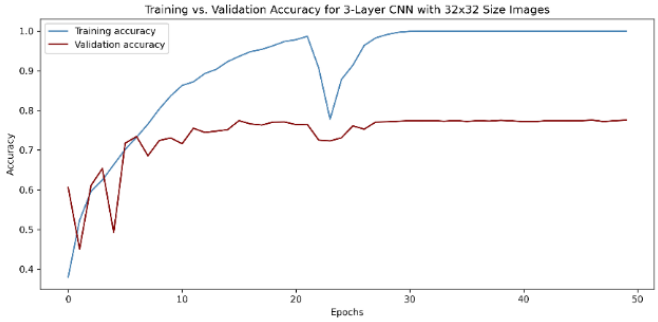

Figure 6: Training vs. validation accuracy for model 5

The lengthy training time of model 3 prompted us to revert to the previous 64x64x1 input shape in model 4 and scale the network down accordingly. Model 4 trained for the same number of epochs as model 3 and its performance results were encouraging. Model 4 reduced the training time of model 3 by 92%, while only

decreasing accuracy by 1.2%. This was a preferred trade-off, and a significant reduction in training time was desired. Curious about the effects of reducing the input size further, we scaled down the images to a 32x32x1 input size in model 5, with the same downscaling of the model as conducted in model 4. Observing the results for model 5 in Table 3, the effect of scaling the images down to such a reduced image size had a negative effect on model 5's performance. This is also displayed in training vs. validation accuracy plots of both models. Figure 5 plots model 4's training vs. validation accuracy curve. In Figure 5, the validation curve closely follows the training curve through all of model 4's 10 epochs. In Figure 6, model 5's training vs. validation accuracy curve shows a significant deviation from the training curve, visually displaying the issues with model 5.

Furthermore, model 5 trained for a total of 50 epochs to boost its accuracy, which was only 77.59%. This eliminated any training time reduction desired by reducing the input size, with only a modest reduction of 5.56 minutes of training time over model 4 but a 16.38% reduction in accuracy. This significant performance loss can be attributed to the severe reduction in image size. This reduced image size lost many unique class features, contributing to the inter-class confusion in similar classes. In model 5's confusion matrix, the similar classes Allaple.A and Allaple.L appeared to the network as the same class. Any feature that could uniquely identify these classes was lost when reducing the image size to 32x32x1. This confusion is displayed visually in Figure 7 which plots model 5's confusion matrix.

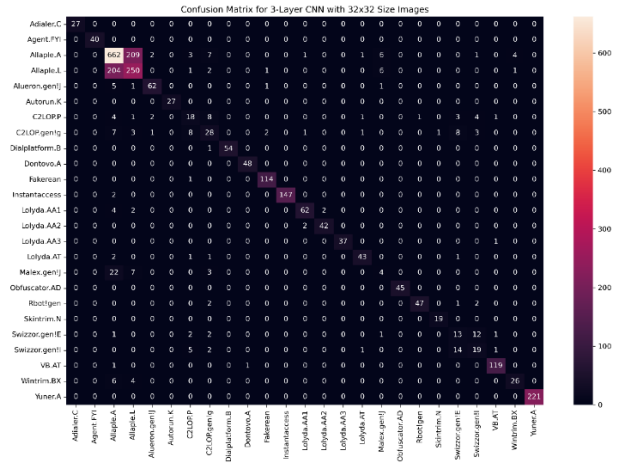

Figure 7: Model 5 confusion matrix

9. Conclusion

Some issues regarding the future viability of this method were not covered in this work. An infinitely complex feature set is the most prescient issue concerning automating malware classification. Since unique malware signatures continue to grow, this will add more classes to an already large dataset. This complexity could prevent learning and cause severe inter-class confusion when presented to a learning model. This confusion could lead to misclassification, which can severely hinder the accuracy of a trained model. The dataset we have utilized is only an extremely small subset of the millions of relevant malware classes currently distributed online. To realistically classify malware of all relevant malware signatures, rather than a small subset, future research must be conducted into generalizing the classes to a level conducive to effective and fast learning.

Another relevant issue that must be addressed is the difficulty of gathering a large malware dataset, which is even more challenging when gathering a large dataset with malicious and benign files. This data aggregation issue does not only affect our research, but it affects all malware researchers. For example, our dataset was not as easily accessible as one might expect, and it did not contain the original malware binaries. This may be due to file hosting services, such as Dropbox, automatically flagging files containing malware. We would have liked to obtain a similar dataset that contained the original PE files, where we would have conducted our own conversion to grayscale images. The PNG images in the Malimg data set were not true grayscale images, having a color depth of 3, rather than 1. This had to be addressed during preprocessing of our dataset, where all the images were converted to true grayscale using Python's image library. Until malware analysts and researchers have an open and easily-accessible dataset to conduct experiments with, this area of research will continue to fall behind other fields of machine learning.

The final issue which must be considered is performance. Training a convolutional neural network is a computationally-intensive process. For example, our third model required nearly four hours of training on a mid-level laptop with no GPU acceleration. Ideally, we would like to run a production version of this automated system on a network edge device. This would be the most conducive application of our proposed malware detection system since static malware analysis techniques are the first line of defense against malicious threats. Edge devices typically have limited computational resources. Providing a neural network that can run natively and in real-time on these edge devices is essential to achieving this goal.

Finally, what has been presented should not be considered a method to supersede or replace any of the current static malware analysis methods. However, the proposed method should be considered a novel tool in a malware analyst's skill chain. Unlike natural language processing, machine learning is not a silver bullet that solves all cybersecurity-related tasks. Instead, machine learning ought to automate a security professional's most time-consuming and repetitive tasks. Automating some of the initial stages of static malware analysis would allow malware analysts more time to gain deeper insights into the malicious file at hand. We are confident that our research shows that our proposed automated static malware analysis method can become a viable real-time malware detection method.

Acknowledgement

This material is based upon work supported by the National Science Foundation under Grant No. 1753900.

References

Anwar, A. (2021). *Difference between Local Response Normalization and Batch Normalization*. [online] Medium. Available at: http://towardsdatascience.com/difference-between-local-response-normalization-and-batch-normalization-272308c034ac [Accessed 3 Feb. 2022].

Bhodia, N., Prajapati, P., Troia, F. and Stamp, M. (2019). *Transfer Learning for Image-Based Malware Classification*.

Gibert, D., Mateu, C., Planes, J. and Vicens, R. (2018). Using convolutional neural networks for classification of malware represented as images. *Journal of Computer Virology and Hacking Techniques*, 15(1), pp.15–28.

ISC2 (2021). *2021 Cybersecurity Workforce Study*. [online] *www.isc2.org*, ISC2, pp.24–25. Available at: https://www.isc2.org//-/media/ISC2/Research/2021/ISC2-Cybersecurity-Workforce-Study-2021.ashx [Accessed 4 Feb. 2022].

Kaspersky Lab (2021). *The number of new malicious files detected every day increases by 5.2% to 360,000 in 2020*. [online] www.kaspersky.com. Available at: https://www.kaspersky.com/about/press-releases/2020_the-number-of-new-malicious-files-detected-every-day-increases-by-52-to-360000-in-2020 [Accessed 4 Feb. 2021].

Le, Q., Boydell, O., Mac Namee, B. and Scanlon, M. (2018). Deep learning at the shallow end: Malware classification for non-domain experts. *Digital Investigation*, [online] 26, pp.S118–S126. Available at: https://www.sciencedirect.com/science/article/pii/S1742287618302032 [Accessed 30 Jan. 2020].

Mallet, H. (2020). *Malware Classification using Convolutional Neural Networks — Step by Step Tutorial*. [online] Medium. Available at: https://towardsdatascience.com/malware-classification-using-convolutional-neural-networks-step-by-step-tutorial-a3e8d97122f.

Microsoft (n.d.). *An In-Depth Look into the Win32 Portable Executable File Format, Part 2: Figures*. [online] bytepointer.com. Available at: https://bytepointer.com/resources/pietrek_in_depth_look_into_pe_format_pt2_figures.htm [Accessed 4 Feb. 2022].

Nataraj, L., Karthikeyan, S., Jacob, G. and Manjunath, B.S. (2011). Malware images. *Proceedings of the 8th International Symposium on Visualization for Cyber Security - VizSec '11*.

Raff, E., Barker, J., Sylvester, J., Brandon, R., Catanzaro, B. and Nicholas, C. (2017). *Malware Detection by Eating a Whole EXE*.

SonicWall (2021). *2021 SonicWall Cyber Threat Report*. [online] *https://www.sonicwall.com/*, Milpitas, CA: SonicWall Inc., p.58. Available at: https://www.sonicwall.com/resources/white-papers/2021-sonicwall-cyber-threat-report/ [Accessed 4 Feb. 2022].

Theta432 (2020a). *Theta432*. [online] www.theta432.com. Available at: http://www.theta432.com/post/malware-analysis-part-1-static-analysis [Accessed 3 Feb. 2022].

Theta432 (2020b). *Theta432*. [online] www.theta432.com. Available at: http://www.theta432.com/post/malware-analysis-series-part-2-dynamic-analysis [Accessed 3 Feb. 2022].

Theta432 (2020c). *Theta432*. [online] Theta432.com. Available at: http://www.theta432.com/post/malware-analysis-series-part3-memory-malware-analysis.

Véstias, M.P. (2019). A Survey of Convolutional Neural Networks on Edge with Reconfigurable Computing. *Algorithms*, 12(8), p.154.