AFRL-RY-WP-TR-2019-0121

NEURAL CLASSIFICATION OF MALWARE-AS-VIDEO WITH CONSIDERATIONS FOR IN-HARDWARE INFERENCING

Michael L. Santacroce

University of Cincinnati

JULY 2019
Final Report

AIR FORCE RESEARCH LABORATORY
SENSORS DIRECTORATE
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7320
AIR FORCE MATERIEL COMMAND
UNITED STATES AIR FORCE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
July 2019	Thesis	17 March 2019 – 26 April 2019

4. TITLE AND SUBTITLE
NEURAL CLASSIFICATION OF MALWARE-AS-VIDEO WITH CONSIDERATIONS FOR IN-HARDWARE INFERENCING

5a. CONTRACT NUMBER
1014236

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
N/A

6. AUTHOR(S)
Michael L. Santacroce

5d. PROJECT NUMBER
N/A

5e. TASK NUMBER
N/A

5f. WORK UNIT NUMBER
N/A

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Cincinnati
2600 Clifton Avenue
Cincinnati, OH 45221

8. PERFORMING ORGANIZATION REPORT NUMBER
AFRL-RY-WP-TR-2019-0121

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory
Sensors Directorate
Wright-Patterson Air Force Base, OH 45433-7320
Air Force Materiel Command
United States Air Force

10. SPONSORING/MONITORING AGENCY ACRONYM(S)
AFRL/RYWA

11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)
AFRL-RY-WP-TR-2019-0121

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
PAO case number 88ABW-2019-2245, Clearance Date 8 May 2019. Report contains color. In Partial Fulfillment of the Requirements for the Degree, Master of Science in the Department of Electrical Engineering and Computer Science.

14. ABSTRACT

The objective of this thesis is to explore the classification of assembly code as benign or malicious through the use of neural networks, and while building these networks, giving consideration to the creation of malware detecting hardware. Neural networks have become a go-to solution in many fields due to their ability to learn from an enormous number of features. Fully entrusting security to a neural network may be unwise due to issues with bias in training data and the ultimately unknowable nature of how the network makes a classification. If a proficient system is achieved for low cost in terms of memory or time, however, it could be another tool in the toolbox for fighting malware.

15. SUBJECT TERMS
Convolutional neural networks, deep learning, FPGA, malware detection

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT:	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON (Monitor)
a. REPORT	b. ABSTRACT	c. THIS PAGE			Daniel Koranek
Unclassified	Unclassified	Unclassified	SAR	108	19b. TELEPHONE NUMBER *(Include Area Code)* N/A

NEURAL CLASSIFICATION OF MALWARE-AS-VIDEO WITH CONSIDERATIONS FOR IN-HARDWARE INFERENCING

A Thesis
Presented to
The Academic Faculty

By

Michael L. Santacroce

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
Department of Electrical Engineering and Computer Science

University of Cincinnati

March 2019

NEURAL CLASSIFICATION OF MALWARE-AS-VIDEO WITH CONSIDERATIONS FOR IN-HARDWARE INFERENCING

Approved by:

Dr. Rashmi Jha, Advisor
Department of Electrical Engineering and Computer Science
University of Cincinnati

Dr. Ali Minai
Department of Electrical Engineering and Computer Science
University of Cincinnati

Dr. Temesguen Kebede
Department of Electrical and Computer Engineering
University of Dayton

Date Approved: March 13, 2019

It’s so hard to remember what you’re doing until it’s done.

bill wurtz

ACKNOWLEDGEMENTS

There are truly too many people to thank. I'm sure that no matter how many people I mention I will leave someone out, however, I'll do my best to share my gratitude.

Thank you to Rashmi Jha for challenging me, helping me grow, and showing me how rewarding research can be. Her research team also helped me crystallize my thoughts and gave useful suggestions throughout my time working. Thanks also go to those from the AFRL - David Kapp, Daniel Koranek, and Temesguen Kebede - for great discussions about the work. Special thanks go to Daniel, for creating the grant proposal, choosing to work with us, and driving to UC so many times to discuss the project. Additional thanks go to Tem and Ali Minai for being on my thesis defense committee, and anyone who was part of the larger project, John Musgrave and Jeremiah Greer especially.

I don't think I could have stayed sane without the support of my wonderful girlfriend Lillian Ashworth, who brought me joy even on the worst days. My friends also helped greatly. Zach Collins specifically kept up my spirits as he embarked on this odyssey at the same time as myself. Those in the IEEE office (that ACM, ACM-W, and Eta Kappa Nu also happens to use) were a constant source of support.

Chris Huynh and Taylor Rowekamp aided in the deployment of the network to the FPGA through debugging and keeping things light as I became progressively more stressed and sick throughout the year. Laura Tebben came up with the recalculation edge case for the Node-Distance algorithm, for which I am grateful, as I was pretty stuck.

Then, of course, thanks goes to my mother Doran and my sisters Alyssa and Calysta for always being there. Even more, they are always proud of me for everything I do. For that, I am filled with determination and appreciation.

This work was supported by National Science Foundation under ECCS 1556294, CNS-1556301 Research Experiences for Undergraduates supplement awards, and AFRL GRANT No. 1014236 under subaward No. 1919.05.05.91 with TDKC.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The objective of this thesis is to explore the classification of assembly code as benign or malicious through the use of neural networks, and while building these networks, giving consideration to the creation of malware detecting hardware. Neural networks have become a go-to solution in many fields due to their ability to learn from an enormous number of features. Fully entrusting security to a neural network may be unwise due to issues with bias in training data and the ultimately unknowable nature of how the network makes a classification. If a proficient system is achieved for low cost in terms of memory or time, however, it could be another tool in the toolbox for fighting malware.

Our approach revolves around turning the executable bytes of code into a video for classification methods, what we call Malware As Video, while relying on minimal preprocessing. We will show that a distributed convolutional network achieves the best performance of the architectures we discuss, hinting that the temporal dependency of the input code may not be important for classification. To do so, we will walk through the attempted architectures and discuss their strengths and weaknesses before arriving to the final network.

Our best network achieves 99.96% training and 99.31% testing accuracy on a dataset of malicious vs. benign code, with an average time to predict of $\sim$8.08ms. It can classify 100% of malware with a false positive rate of 13%. We also test the network in a few distinct ways. The network achieves 98.47% accuracy on a 0-Day malware simulation. The network was also given only partial inputs, achieving around 90% accuracy when using only 20% of each input file on a dataset of 9 classes of malware.

We also perform pruning and quantization strategies on the distributed convolutional network so it can be as small as possible, enabling a wider range of systems the network could operate on. We use a novel pruning technique we call Node-Distance Pruning and we then use Jenk's Natural Breaks for determining quantization buckets. The network is finally deployed to an FPGA as a proof-of-concept chip for malware detection.

CHAPTER 1
INTRODUCTION

1.1 Scope

Computers are an essential hazard to life in modern society, providing a seemingly unlimited wealth of knowledge, entertainment, and risk to us all. Visiting the wrong website at the wrong time can result in losing personal data, bank accounts being stolen, or worse, jeopardizing one's safety and livelihood. The never-ending arms race with malware is no trivial matter; for consumers, for businesses, for governments, or even for hospitals [1], which can all be the victim of indiscriminate or targeted attacks. Malware prevention begins with malware detection as it is vital to know if a system has become infected. Hundreds of methods for detecting malware have been developed but it is important to continue development as each method eventually becomes antiquated.

At the same time, biologically inspired neural networks have boomed into a massively popular area of research spanning all fields of science and technology. Their widespread influence, spurred on by seemingly never-ending developments in the past 60 years, begs a natural question in respect to the malware arms race: how can a neural network be used in the fight against malware? Even more specifically, how can we safely and efficiently use a neural network to fight malware?

The work presented here investigates the possibility of using neural networks for malware identification, with practical considerations to the size, usage, and deployment of a neural-network-based, malware-detecting subsystem. We base our work on using the raw executable assembly code of programs as the input to a neural network. Using assembly code as an input allows for malware detection at the lowest possible level, where no backdoor through the operating system could be exploited, opening the possibility of a malware

detecting hardware based on neural networks. Our hypothesis is that putting this limit on the input can result in equivalent results to seminal works while enabling leaner and faster networks.

Building a full system with an integrated malware detecting chip would require more work than simply plugging a custom card into a motherboard so we primarily investigate this by simulating the networks and their input. We also conduct an attempt at deploying the most successful network to an FPGA to evaluate the feasibility of such a system.

1.2 Outline

Chapter 2 first overviews some prerequisite concepts in neural networks such as their basic elements, notation, and neural network compression. Then, a short review of seminal literature based on malware detection through bytecode is conducted, comparing major points from these works to our own.

In Chapter 3 we discuss the methodology of obtaining data, investigating neural network architectures, compressing the final network, and deploying the compressed network to an FPGA. We will discuss our central idea, MaV, and networks that can be built off of this input. We also introduce a novel technique, Node-Distance Pruning, which empirically works well to slim down our network.

After discussing the methodology, Chapter 4 first presents the results of the attempted neural network architectures to justify claims made in Chapter 3. The best neural network is then put through a myriad of tests to evaluate its capabilities in malware detection in comparison to seminal works.

Chapter 5 finally concludes our work, summarizing the contributions made and how future work can expand on those contributions.

CHAPTER 2

BACKGROUND

2.1 Topic Areas

We will cover some prerequisite topics in this chapter so that the reader has an understanding of work we present later on.

First, we will overview concepts in malware detection and neural networks. We will then perform a review on seminal literature concerning the classification of bytecode with neural networks, looking at strengths and weaknesses of the various methods for later comparison.

2.1.1 Malware Detection

Malware detection is a vast field with a wide array of developed techniques [2, 3], and even within the field of malware detection there is no shortage of methods that use neural networks, such as work using high-level features [4]. The most important concept from malware detection is that methods are largely based on one of three techniques: static analysis, dynamic analysis, or a hybrid of the two. Static analysis uses feature of malware outside of its execution to make a classification while dynamic analysis focuses on the features of a program during execution to make a classification. An example of static analysis might be analyzing the opcodes used in a candidate program, where dynamic analysis might focus on runtime-exclusive information, such as the systems used by the program. Static analysis can often be avoided through obfuscation [5], though obfuscation counter-techniques have been developed in tandem with attacker developments [6, 7]. Dynamic analysis is not impervious to avoidance either [8], contributing to the never-ending race of malware detection.

All work presented in this thesis is therefore a form of static analysis, though it is not necessarily limited to static analysis. Our datasets consist of bytecode pulled from executables without any kind of runtime, however, it might be possible to use the live commands run by a CPU as a replacement to the static input.

2.1.2 Overview of Neural Networks

We assume that the reader has some familiarity with the general concepts of neural networks, however, we will go over some important concepts here. This section is not meant to be an encompassing work on neural networks.

A standard neuron i takes the form shown in Figure 2.1, where the neuron takes the summation of its weights multiplied by the corresponding inputs and then applies its activation function.

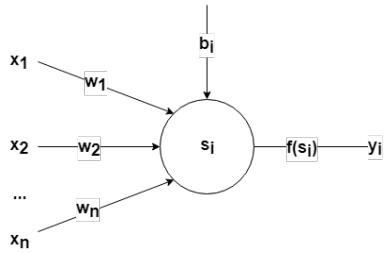

Figure 2.1: Standard Neuron

Where x_j represents part j of the input, w_{ij} represents the weight from x_j to neuron i, b_i is the bias for neuron i, and f_i represents the activation function for the neuron. Common activation functions we will use include the sigmoid function, Rectified Linear Unit, and hyperbolic tangent. The predicted output $\hat{y}_i$ of neuron i given input q can then be written as:

$$\begin{aligned} s_i^q &= \sum_{j=0}^{n} w_{ij} x_j^q + b_i \\ \hat{y}_i^q &= f_i(s_i^q) \end{aligned} \tag{2.1}$$

The output can also be rewritten simply as a dot product in matrix form $s_i^q = w_i \cdot x^q + b_i$.

A helpful way to view a neuron is a way of mapping an input feature space to a new feature space determined by the activation function. In other words, the neuron draws a hyperplane in the feature space. Then, using hyperbolic tangent as an example activation function, any inputs falling on one side of the hyperplane evaluate to -1 and any inputs falling on the other side the hyperplane evaluate to 1. The important feature of the activation function is that it is nonlinear; if it were not, networks of neurons would simply collapse into a linear combination of inputs.

We have been calling it neuron i to set up that many neurons can be used in tandem to form what is called a *layer* with other neurons. Neurons form a layer if they are all use the same input vector. Their outputs can then be considered as one matrix.

Neurons have become popular due to their ability to be trained in a supervised manner with gradient descent and its variations. To train a layer of neurons, a loss function J is necessary (also known as a fitness, objective, or error function). The loss function essentially puts a number to how incorrect the neuron is given an input q. A straightforward loss function is Mean Squared Error (MSE) from all inputs:

$$J^q = \frac{1}{2}\sum_{i=1}^{l}(y_i^q - \hat{y}_i^q)^2 \tag{2.2}$$

Where l is the number of neurons in the layer, y_i^q is the desired output for neuron i given input q, and $\hat{y}_i^q$ is the actual output of the network for neuron i.

The loss function we will use is categorical crossentropy, defined as:

$$J^q = -\sum_{c=1}^{M} y_c^q \log \hat{y}_c^q \tag{2.3}$$

Where M is the number of classes, y_c is the desired output class label, and $\hat{y}_c$ is the predicted probability from the network. Unlike MSE, this loss function will function as a way to output probabilities for two or more classes given an input, which is especially useful for multiclass classification. For example, there may be nine neurons in the layer for

nine classes of data, to predict the likelihood of an input belonging to each class. Another option might be to have only one neuron as a binary "yes" or "no" output.

With a loss function chosen, the layer of neurons then can be trained using a nonlinear optimization method which is generally gradient descent. Gradient descent is a greedy algorithm that operates by iteratively taking steps on the surface of the error function towards the closest local minimum with the goal of finding the global minimum. When the loss function is minimized, the network is getting the most predicted classes correct.

Gradient descent requires the partial derivative of the loss function to each parameter (weigh or bias) with respect to input q. These partial derivatives can be found by repeatedly applying the chain rule. For an arbitrary loss function J and standard neuron i given input q:

$$\frac{\partial J^q}{\partial w_{ij}} = \frac{\partial J^q}{\partial \hat{y}_i^q} \cdot \frac{\partial \hat{y}_i^q}{\partial s_i^q} \cdot \frac{\partial \hat{s}_i^q}{\partial w_{ij}} \tag{2.4}$$

Where each of these partial derivatives can then be found for the standard neuron as:

$$\frac{\partial J^q}{\partial w_{ij}} = \frac{\partial J^q}{\partial \hat{y}_i^q} \cdot f_i'(s_i^q) \cdot x_j^q \tag{2.5}$$

Where $\frac{\partial J^q}{\partial \hat{y}_i^q}$ depends on the loss function and $f_i'(x)$ is the derivative of the activation function. By design, both of these derivatives are generally easy to find.

To train a layer of neurons, input q is presented and the error calculated with the loss function. Then, the partial derivative can be calculated for each weight and bias using the equations as shown above. Gradient descent then uses those partial derivatives to take steps towards local minimums in the loss function.

Until now, we have been discussing the structure and training of a layer of neurons - what makes these layers so useful and popular is their ability to be stacked. A *neural network* consists of multiple layer of neurons, where the output of one layer is used as inputs to the next layer of neurons. Layers of the network that are between the input and

output are called *hidden layers*.

When multiple layers of neurons are used to form a neural network, taking the partial derivative of the loss function with respect to a parameters is less straightforward. To do so, we use the process of *backpropogation*, which gains its name from the process of propogating the error backwards from the output of the network backward to all parameters. Backpropogation generally entails more applications of the chain rule.

Gradient descent usually occurs in Epochs, or passes through the entire dataset. Often times data is presented in batches so the network is training on, for example, five inputs instead of one at a time. We do not perform batch training due to limitations with the libraries used. Many variants on gradient descent exist for different purposes and to address various problems in training [9]. Throughout our work we generally use RMSProp, which adjusts the learning rate while training in an attempt to avoid skipping over minima [10].

Neural networks do not need to take the form of connecting each input to each neuron. In fact, neurons can perform virtually any operation as long as it is differentiable. Another, popular neural network architecture is the Convolutional neural network, which borrows 2d convolution from fields like image processing. Convolution can be viewed as "sliding" a small filter over an image of pixels. While the filter slides, every overlapping number is multiplied and then the outputs summed to result in the final output pixel. An example is shown in Figure 2.2.

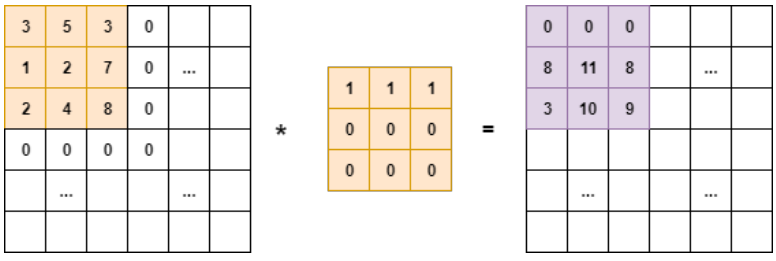

Figure 2.2: Convolution Example

Figure 2.2 shows how the filter is multiplied by the input image and the output summed to make the output pixel. Convolution usually results in an image smaller than the input

image, however, the output can also be kept the same size by imagining the input image padded with 0's. Doing so enables deeper networks that would otherwise reduce the output size so one value.

Implementing neural networks has been streamlined thanks to libraries built around symbolic differentiation, linear algebra optimization, and GPU usage. Before these tools it was necessary to calculate the error with respect to the loss function for a given parameter manually, but with them, the error is automatically calculated using symbolic differentiation. In essence, the operations are transformed into a graph composed of nodes and edges so that the derivatives can be algorithmically determined. Optimized code can then be created with custom linear algebra operations, reducing complexity and utilizing hardware like GPUs (or FPGAs, which we will discuss later on).

Throughout this work, Keras is used extensively [11]. Keras makes network building and prototyping quick and painless by allowing the creation and training of a network in a couple dozen lines of code. Many common layers are pre-defined, such as densely-connected layers or convolutional layers. Adding custom layers or training methods is also easy due to the open-source nature of the library and available templates in the documentation. The library does not build the symbolic graphs itself, but instead can use multiple backends, of which we use Tensorflow. Layer on we will use a different version of Tensorflow's own backend, XLA [12], which we discuss in-depth in Chapter 3.

2.1.3 Convolutional, Recurrent Neural Networks

One attempted network architecture that achieved some success was that of convolutional, recurrent neural networks. Here we will go through what it means for a network to be both convolutional and recurrent for reference material later on. To simplify the notation, we will no longer designate a specific input q and assume any equation given is for a given input sample.

Recurrent neural networks have been a staple in classifying time-dependent data for

many years. These networks can be seen as a system with feedback, or a system with internal state, and they take many forms for many various purposes. Perhaps the most popular recurrent neural network is the LSTM or Long Short Term Memory network, with equations shown in 2.6.

$$
\begin{aligned}
i &= \sigma(h_{t-1} \cdot U_i + x_t \cdot W_i) \\
f &= \sigma(h_{t-1} \cdot U_f + x_t \cdot W_f) \\
o &= \sigma(h_{t-1} \cdot U_o + x_t \cdot W_o) \\
g &= tanh(h_{t-1} \cdot U_g + x_t \cdot W_g) \\
c_t &= f \cdot c_{t-1} + i \cdot g \\
h_t &= o \cdot tanh(c_t)
\end{aligned}
\tag{2.6}
$$

Where σ represents the sigmoid function, $W_{i,f,o,g}$ and $U_{i,f,o,g}$ are trainable weight vectors, and c_t and h_t are the hidden state and output, respectively. Of course, between works, these equations can vary and many studies have been done on the effects of adding or removing components. We present these equations specifically as they correlate to those shown in 2.7. LSTMs have a large number of benefits, one of the biggest being that they can help deal with the *vanishing gradient* issue when training [13, 14], which enables training on much longer time sequences.

While LSTMs are useful on their own they struggle in this particular field due to the high dimensionality and extreme length of the sequences. Instead, convolutional neural networks seem much more suited, especially considering the intersection with image processing and classification. The issue with a vanilla convolutional network, however, is that it can only accept a fixed-size input.

To get the best of both worlds we can use a recent network architecture called the ConvLSTM. Introduced in 2015, the ConvLSTM uses convolution in place of the dot product for a few select operations to capture both temporal and spatial relationships in data [15].

One neuron can be described as shown in 2.7.

$$
\begin{aligned}
i &= \sigma(h_{t-1} * U_i + x_t * W_i) \\
f &= \sigma(h_{t-1} * U_f + x_t * W_f) \\
o &= \sigma(h_{t-1} * U_o + x_t * W_o) \\
g &= \sigma(h_{t-1} * U_g + x_t * W_g) \\
c_t &= f \cdot c_{t-1} + i \cdot g \\
h_t &= o \cdot tanh(c_t)
\end{aligned}
\tag{2.7}
$$

Where σ represents the sigmoid function, $W_{i,f,o,g}$ and $U_{i,f,o,g}$ are trainable weight vectors, and c_t and h_t are the hidden state and output, respectively.

Since their introduction, ConvLSTMs have found use in many applications, of which we will now mention a few. One work uses them specifically to identify temporal relations in gesture recognition after feature extraction with 3d convolution [16]. One ConvLSTM network is used to predict traffic accidents and even build on the architecture by proposing the Hetero-ConvLSTM due to challenges with their dataset [17]. More directly relating to this work, some works use ConvLSTMs for working with sequences of images[18, 19]. Another work uses the ConvLSTM as a nested memory model and again expands on the architecture [18]. One work actually uses a different architecture that they call Long-Term Recurrent Convolutional Networks, however, the goal is the same as the ConvLSTM [19].

2.1.4 Neural Network Compression

Neural Networks are often times massive due to their nature, both in amount of space and length of time required to run them. It is desirable to make a given network as small as possible through various compression methods, namely in this work pruning and quantization. We consider compression of the utmost importance for our work as a malware-detecting neural network is only as useful as the feasibility of its implementation.

One simple question looms over many neural network works: how big does one make

the network? Given the unknowable nature of what features each neuron will look for in a network, what features are even important in a many-dimensional space, or at what point neurons become redundant, it is not easy to simply calculate how many neurons should be in the network. Pruning neural networks is the process of removing the least active connections or neurons in a neural network and retraining the network with the goal of having minimal impact on the overall performance. The advantage of pruning is that there is theoretically some unknowable optimal size of neural network per problem that might be found after a larger network is trained and repeatedly pruned [20]. It is up for debate if pruning a network is only useful as an architecture search in finding the optimal number of neurons, or if pruning a trained, larger network will yield better performance in the end [21, 22]. Regardless, pruning is an important for methodically testing a neural network instead of brute-force guessing at what size may work.

Measuring the importance of a connection or neuron in a neural network is not an easy task, and as such, dozens of pruning methods have been developed [23, 24, 25, 26, 27, 28], ranging from simple rules-of-thumb to complex, mathematical analysis. Often times, choosing a pruning method unfit for a network or using a pruning method too much can break a trained network, rendering it untrainable afterwards. We observed this effect in our own testing many times.

Alongside pruning exists the concept of network quantization. We refer to quantization as the process of grouping the parameters of a neural network by similarity, though it can have other meanings. For instance, in digital signal processing, it often means that every number at every step of a transfer function or some process is put into discrete buckets based on the range of possible numbers that may come out of that process [29]. Methods similar to that exist for neural networks, however, we do not go that far. Instead, we only take the weights and biases (or any other parameters) of each layer in a neural network and group them into bins, resulting in *weight sharing*, similar to other works [30, 31, 32, 33]. Ideal quantization would cause no loss in performance of the network, but if there is a

loss in performance, it is a simple matter of using more bins (or cluster centers) to increase performance. One work compares methodologies for quantizing layers with clustering, especially in comparison to existing matrix factorization methods [30]. Multiple works combine both pruning and quantization in a similar way to our work to show the potential gains that can be had [33, 31].

2.2 Neural Networks for Classifying Executables/Assembly Instructions

Malware detection and machine learning intersect in a plethora of techniques and papers. Our work will center around and ultimately build on what we call Malware-as-Image or MaI, an active research area that converts the bytes of an executable into an image for classification. Creating malware images generally consists of converting each individual byte of an arbitrary bytecode into decimal values (ranging from 0-255) which can then be used as pixels in an image. Parallels from this idea to image processing and classification can then be applied. We will present some works for reference and comparison to our work, showing the state of the art and also showing how our method carries some advantages and disadvantages. Many methods have been developed for MaI classification and we do not consider this an exhaustive survey. We have instead selected specific papers to illustrate specific points in relation to our work.

Before looking at these works there are some vital points to emphasize now that distinguish our work from almost all others: minimal preprocessing and only using the .text section of the code. Programs come at so many varying lengths that choosing how to deal with the input becomes a problem in itself. Machine learning methods like support vector machines or neural networks usually handle fixed size inputs (in their vanilla form), meaning that the data must be processed to that fixed size. Introducing preprocessing to adjust the input to that fixed size also introduces a few issues. Preprocessing can widen margins for error, requires a time and space cost, and possibly even introduces security concerns if the preprocessing is reverse-engineered by attackers. Some authors note that, depending on the methods used, it may be possible for an attacker to figure out some way to exploit preprocessing to hide malicious code. Our work specifically avoids preprocessing as much as possible. To aid in cutting down the size of the input we also only use the .text section, or what should be the actual assembly commands, of the binaries. We will discuss later on some additional benefits of avoiding preprocessing and only using .text, however, we note

these differences now to contrast with other works.

The earliest work we could find that relates to MaI is one that uses Self-Organizing Maps (SOMs) to analyze Windows executables infected with malware [34]. SOMs are a special class of neural networks that are trained in an unsupervised manner having neighboring neurons connected, with the goal of projecting the input into a discrete features space that can be observed. Ultimately, noticeable differences are found between regular Windows code and infected code. Even further, viruses in the same family give similar feature maps. While this approach was not used for large-scale file classification it gives credence to the notion that there are features of malware compared to benign code that are easily detected by a neural network.

Neural networks are by no means the only machine learning method, and other methods have shown success. In 2006, a variety of techniques were used to classify executables, including decision trees and Support Vector Machines through boosting [35]. In the end, decision trees obtained an Area Under Curve (AOC) of 99.6%, which is competitive to results even now. While it achieves great performance, this work utilizes over 225 million n-grams, however, and would face considerable challenges in implementation in comparison to our methods, though the authors do implement a full test of their methods on wild malware.

In 2010, one work was able to achieve 98% accuracy in classifying around 9,500 samples of 25 classes of malware [36]. Their methods involve using feature projection into lower dimensional space and k-NN, ultimately taking 56 seconds to classify a file (though that time to classify was from 2010 and would likely be faster today). The authors note possible methods of avoiding detection, like dummy bytes that are added to a file.

One work uses a feedforward network on extracted features to achieve 96.35% accuracy on 3131 samples of 24 classes of malware [37]. Here the malware images are preprocessed and features extracted with the Gabor+Gist descriptor so that they can be fed to a densely-connected, feed-forward neural network. Given the extremely large number of classes,

performance is excellent, and we reference this work to again give credence to the notion that neural networks are able to identify features in malware, though the network comes after heavy feature extraction.

More recent times have focused much more heavily on Convolutional Neural Networks (CNNs). Because CNNs are able to identify spatial relationships in data and images they are a natural choice to pair with MaI classification [38, 39, 40]. Another big player came in the introduction of the Microsoft Malware Classification Challenge [41], which encouraged users on Kaggle to classify over 20,000 samples of nine classes of malware. Because of the high availability of the data and prominence of neural networks in literature, MaI classification is experiencing high amounts of interest.

In one thesis work, the author uses a straightforward CNN to achieve 95.24% accuracy on a test set of over 40,000 samples of malicious or benign code, followed by a residual network to achieve 98.21% accuracy on the same samples [42]. We use this work as a comparison point, where again the malware images are preprocessed to the same size and the entire file is used. The use of the residual net is an interesting comparison to the convolutional net as well, where residual nets operate by "[...] instead of just hoping that deep nets will divide stack of layers and learn desired mappings better on their own as we increase the depth, we explicitly make a stack of layers learn the mapping" [42].

Our own earliest work obtained up to 88% accuracy on the Microsoft Malware Classification Challenge dataset [43]. The largest issue for performance in this work was that we padded each image to the same size instead of preprocessing them. While no information is lost in this method, large amounts of padding would inhibit performance on smaller files. In this work, however, we introduced the idea that only the .text section was necessary for classification. We have two more unpublished works in which we will pull from throughout this thesis. In these works, we establish the idea of MaV, and develop network architectures to classify binaries based on MaV [44, 45].

Kedebe and Narayanan et al. have also done substantial work in classifying malware as

an image, with particular attention to various classification methods, autoencoders for classification, and working with the imbalanced classes in the Microsoft Malware Classification Challenge dataset [46, 47, 48]. In particular, the authors compare various machine learning methods for malware detection, showing that k-NN can actually achieve greater performance to an ANN and SVM when used on malware images after Principal Component Analysis (PCA) [47]. The authors then extend their work to using an autoencoder-based network for classification [46], noting the possible disadvantages with PCA methods and using only resized malware images as input. An overall performance of 99.15% accuracy is shown alongside full confusion matrices. The authors also note that a CNN would be an interesting area of future work. Finally, the authors combine previous methods to overcome the problem of imbalanced classes as one class in this dataset has a disproportionately low number of samples and was previously excluded in other works [48]. We do not address class imbalance specifically in this work, however, it is an important problem to think about when classifying malware.

The paper that best worked with the idea of code as an image was by one work that uses a CNN and LSTM in an ensemble configuration [49]. Interestingly, they use the LSTM on only the opcodes of the input and the CNN on the full input, resulting in a maximum accuracy of 99.88% on 40,000 samples balanced equally between malicious and benign files. As we will do ourselves, they also test the network on only the nine classes of malicious files, resulting in 99.36% accuracy. Again, the input is the full executable which is preprocessed to a specified length which brings up practical concerns as we have discussed.

Researchers at Nvidia used an embedding, then a convolutional, distributed for hidden neural network layers which fed into an ending recurrent neural network on over 2 million input files [50] This network uses no preprocessing, unlike other methods, and the authors point out issues with preprocessing. This network architecture most closely resembles our own, however, the overall architecture of the distributed nature of the network is different.

Our architecture focuses on the idea of programs as a video, which we will elaborate on in the next chapter. The authors also note difficulties in using RNNs due to the length of the sequence as we also found. This network architecture achieves a maximum of 94% accuracy and 98% AUC. We also expand on this work in a few ways. The largest difference is that we only use the assembly instructions of the executables (and no data), which cuts down on the size of the input dramatically. We also attempt to integrate the convolution with the RNN in the form of applying a ConvLSTM which does alleviate issues with RNNs. Other tweaks are also made to the architecture, such as using global average pooling to improve generalization [51]. And finally, while we are using a much smaller dataset that likely does not represent the input as well, we achieve better overall performance.

Executable code as an image has become a hot area of research in recent years. In many of these works the images are preprocessed before classification which might take a large amount of time or resources in practice depending on the operations, and may be bypassable by reverse-engineering the preprocessing. Additionally, all of the works we have seen use the full executable file for classification. By only using the .text section of an executable, classification can be done at a lower level in the hardware stack, with less memory, less time, and less context of what is running.

One work that has a high amount of overlap with our own comes in the form of what the authors call MAP, a Malware-Aware Processor [52]. In their work, the authors use logistic regression and a densely-connected neural network with one hidden layer to identify malware in built into a soft processor. After achieving sufficient results, the authors deploy the model to an FPGA alongside an open source x86 CPU core. Their work shows low impact on the CPU for the malware-detecting subsystem and good performance in identifying malware. The authors also evaluate multiple types of input alongside assembly instructions, such as memory address, architectural, and branch features. They settle on using an input of the frequency of assembly instructions with the largest difference. Additionally, many optimization methods are used to accelerate the FPGA neural network. The network can

classify all malware with a 7% false positive rate with after-the-fact detection, or 94% of malware with a 7% false positive rate with runtime detection, on a dataset of about 1,500 samples. We expand on their work by using a method that does not need adaptation for runtime detection. The authors adapt their static method to a method that can run on arbitrary length time series data by using an Exponentially Weighted Moving Average, ultimately hurting performance. In comparison, our method is adaptable to live running code with no performance loss due to the architecture. Finally, it is possible that their detection method may be avoidable in a 0-day scenario by purposely including a ratio of useless commands to disguise the file as benign. Such a scenario should not be detrimental to our results.

CHAPTER 3

METHODOLOGY

3.1 Datasets

The input given to any machine learning algorithm can be as important as the algorithm itself. A machine learning algorithm can give an output that only has limited meaning or is possibly even useless without data that accurately reflects the conditions of what it is trying to model. Throughout this work, the data given to the neural network models evolved as the models did, and not every model was tested on every dataset as new information was learned. We will therefore go through the different datasets used and then explicitly label them to avoid any confusion.

Our largest limitation is obtaining data to use. Ideally, these samples would be obtained by running malicious code on a system and capturing the machine code run on the CPU. It would be extremely difficult, however, to obtain and run many thousands of malicious (or even benign) files and capture that information. Instead, we use the bytecode of executables under the assumption that it is a similar problem.

3.1.1 Common Dataset Elements

Before going through the various sets, there are a few distinctions that can be made across the board. Firstly, all data used in this work will only use the .text section of the executables, which should be the actual code of the executable. There is no standard number, but these sections are likely to be a small portion of the full size of the executable, therefore, using only the executable commands section of the data cuts out a large portion of the input. If a network is able to classify malware based on less information without loss of performance it would be advantageous compared to networks that do not due to the compu-

tational and memory savings. Additionally, we assume that if the network is to be deployed commercially, it should be much easier and more secure to classify code based only on the code itself rather than the code plus additional information. For example, if the network is deployed as a process in the operating system and requires the header information of executables it may be possible for an attacker to hack the operating system itself and trick the network. On the other hand, if the network is directly integrated into the hardware physically, the wiring would need to be physically altered to trick the network. Such a deployment is left for future work, however, we argue that doing the same job with less information is advantageous to requiring more information.

It is important to recognize that .text might contain more than assembly code, or that assembly code can exist outside of the .text section. Malware can easily hide code throughout the executable as a way of disguising itself from attackers. Regardless, we argue that we find competitive results can be achieved from only using the .text section, whatever that section may contain. Using only this section is a means to an end and would not be done in final implementation; in practice, the goal would be to port the assembly commands about to be run by a CPU into the network to check for malicious behavior.

Other papers we have mentioned only use the opcodes of the assembly instructions rather than the opcodes with arguments, which we use. We empirically found that using opcodes alone resulted in a slightly worse performance regardless of network architecture. Of course, using only opcodes would significantly reduce the input size and time-to-predict. To achieve the best performance we use the arguments along with the opcodes, however, it is by no means necessary and our methods could be used either way.

Another process used for all data is randomly pooling 9 of every 10 lines of data. The process is performed by taking every 10 lines and choosing one line to keep of the group, then moving to the next group. Pooling is performed simply to reduce the amount of time and memory it takes to train and predict on data, and in certain cases even improves the testing performance. Unfortunately, the pooling is admittedly the largest flaw in our methodol-

ogy, however, we see no negative difference in performance on pooled vs. unpooled data, as will be shown in the results section. It may theoretically be possible to avoid detection based on the random pooling but further study would have to be done. More importantly, the random pooling would add some amount of time to preprocessing, which we often cite as and advantage to other methods. While there would be some preprocessing involved, this time would be minimal or unnoticeable in practice, consisting simply of one if-statement, as opposed to other preprocessing methods that perform computation on the images.

We have looked at many works that focused on classifying malware as an image. One distinction that is sometimes not discussed is whether each line is padded to a standard length or if the bytes are treated as one contiguous stream of bytes. Figure 3.1 shows an instance of padding for each line, and Figure 3.2 shows an instance of treating the code as a contiguous block. Additionally, in 3.2, there is a chunk of padding at the end of the image, which is used to standardize the images to either one length or a multiple of lengths as will be explained shortly.

Figure 3.1: Malware Image with Line Padding (from [43])

Figure 3.2: Malware Image without Line Padding (from [45])

We began our network architecture search using the line-padding technique as it was

more intuitive to use as each row of the resulting image begins with the opcode. One may theorize that with the opcode in the same place every time, it may be easier for a neural network to recognize the opcode patterns compared to the malware class. In practice, however, we found that the network achieved better results without padding likely because the padding added a substantial amount of useless information to the images.

Additionally, for some networks, the input would be too large for the system used to handle with line padding. To combat the size of the files we imposed a maximum length of 50,000 lines which covers a large majority of input files. This limit was imposed as a temporary measure to begin using networks on the input files. As the network architectures became slimmer and faster, however, we were eventually able to use all files without a line limit. This line limit is, of course, undesirable, but important to note as it was a limitation with the older network architectures.

At the end of this chapter, we will create a table of all datasets used that may be referred to later on.

3.1.2 Naive Image Approach

Our first work was largely to replicate other works and expand on them slightly by only using the .text section of code [43]. This first dataset was the most naive and consisted of 120 samples of malware from the website dasmalwerk.eu [53], and 120 samples of benign code from the C: drive of a default windows installation, split into 160 training and 80 testing samples. The samples were then disassembled using the linux utility objdump. Because objdump outputs the assembly code (such as add, sub, mov, etc), the code must then be broken into tokens and assigned numbers.

The following is a section from the paper we wrote for NAECON 2018 [43]:

“Once the input is disassembled, the next step was to break up the input into arrays of tokens so that they can be formatted for input into the network. Custom automata were used for this process. Words such as add, sub, or eax, were separated by themselves, and

individual characters such as +, [, or * that merit their own tokens were also separated out. Commas were completely discarded, as were comments and whitespace. Assembly also contains hexadecimal numbers for many purposes and unique tokens for procedure names, both of which were all replaced with two different tokens respectively. Here is a difference between assembly and machine code; in the latter, it would be difficult to detect such tokens as everything is a hexadecimal number. In assembly, these tokens must be replaced so the next step of assigning numbers to tokens can function (there are infinite permutations of procedure names as they are arbitrary, and hex numbers can mean anything from an object location to a real number, which introduces ambiguity). These replacements did not seem to have a large impact on the performance of the network positively or negatively as later results suggest using raw machine code is possible.

Following tokenization, each unique token receives a random integer value in a process nicknamed vectorization. The total size of the vocabulary was measured at 903 unique tokens and any newly encountered token was randomly assigned an integer between 1 and 903. A vector is created per file that is padded with 0s to all be the same size. The resultant vector per file was 107071 lines of 25 tokens per line."

This created the first benign vs. malicious dataset. We performed the same process on files from the Microsoft Malware Classification Challenge dataset, resulting in a vocabulary size of 202. For this dataset, we did impose a line limit of 50,000 lines for the work done in [43]. The line limit in this case was not to fit the dataset in memory, but to increase performance. Large amounts of excessive padding would inhibit performance, especially given that the dataset is largely biased towards smaller files. Additionally, only 400 training and 100 testing samples were used from this dataset to increase speed of training.

This technique of using objdump and then vectorizing the resultant code is complex and hard to replicate, not to mention impractical for deploying the network. Additionally, the malware vs. benign dataset itself is extremely limited by only being 240 samples, and the Microsoft Classification Challenge dataset limited to only 500. Again, these datasets

were a proof-of-concept to get the project going and have since been archived.

3.1.3 Malware as Video (MaV)

The largest issue that malware classification faces, as discussed, is the highly variable length of the input. An input program has a minimum length of perhaps even a couple lines, and a theoretical maximum length of infinite lines. How, then, can all programs be classified as malicious or benign with one neural network? As discussed, nearly all methods preprocess the input to a standard length for classification, which introduces computation time and possible information loss.

Our work is largely based on the novel idea of formatting an unknown input code as a video rather than an image, which we call Malware as Video or MaV for short. The format can be easily explained as shown in Figure 3.3.

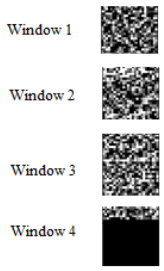

Figure 3.3: Malware Video

In Figure 3.3, we show an example malicious image being formatted as a video. Every file is padded to be an integer multiple of the window size to ensure that it can be broken up. We will refer to *frames* and *windows* as synonymous terms, both meaning one timestep of a malware video with a standardized size. The analogy of malware as a video is not perfect, as frames in a video are largely composed of similar information to the frame before. For instance, if a man is running across the screen in a video, the background will change only minimally. In this case, we split the code into completely different sections with no overlap. If viewed, this malware video would look like static on a T.V. screen. Overlapping each frame by a certain amount of lines could alleviate this discrepancy, however, we found no

difference in practice when overlapping was performed as shown in the next chapter.

By far, the most used dataset for our work comes from the Microsoft Malware Classification Challenge on Kaggle [41]. As stated in the paper, this dataset contains over 20,000 samples of nine classes of disassembled malware and has since become a benchmark for malware classification.

We use this work for every architecture presented as our main dataset for two reasons. Firstly, using only one dataset from one sources ensures no bias and compatibility. There are some concerns with mixing these files with benign files for a malicious vs. benign comparison, largely because we cannot ensure that the conditions of disassembly exactly match that done by Microsoft. Additionally, classifying each file out of nine types of malware can be seen as a similar, if not harder, problem to classifying a file out of two classes, benign or malicious. We assume that some malware classes likely share features with other malware classes in some ways, possibly more so than a blanket comparison of malware to benign. In some cases it can be hard to call a program as benign or malicious - some programs can be borderline at best. Therefore, while this dataset may not include benign bytecode, it is valid to claim that a network that works on classifying this data should also work for a benign vs. malicious dataset.

Of the files from the Microsoft Malware Classification Challenge, 10,869 had labels, and 10,375 samples were found to have .text sections that could be extracted from the file. Therefore, for the full dataset, 8,000 files were used for training and 2,375 were used for testing. We did not process files that did not have extractable .text sections as we are attempting to simulate a network that works on assembly commands from a CPU. These methods would not work for an in-software system as an attacker could simply put malicious code in another section. Since so few files did not have a .text section we did not consider these to bias the results.

As discussed previously the input must be limited to files under 50,000 lines for some networks, resulting in 8,550 total files, of which we used 6,500 training and 2,050 for

testing.

While there are numerous advantages to only using the Microsoft Malware Classification Challenge dataset, to ensure that the networks did indeed classify malicious vs. benign code, we also created a dataset of 6,015 disassembled executables from Windows, obtained from the AFRL. There are a few concerns we have with this dataset to discuss. The following is an excerpt from a submission to [45], a yet unpublished work:

"While we are comfortable that the datasets are now fair to compare we will show those concerns and how we addressed them. Many benign files were initially identifiable by eye due to some certain characteristics, as in [Figure 3.4]:

Figure 3.4: First 100 lines of a benign code image

The first problem is that there seems to be a long portion of data bytes at the beginning of many benign files that only includes one or two executable bytes. In the Microsoft dataset, these sections are grouped to have many data bytes per line (up to 18). Next, many lines end in the same color pixel. This is because of how we extract data from the disassembled bytecode - once disassembled from IDA, we simply look at every line to see if it starts with ".text", signifying executable code. Then, we extract the hexadecimal bytes from that line until there are no bytes left. Many of these lines end in the byte "db", a signal from IDA that represents the variable size (a byte) and not code. For some reason these extra "db" tokens were never an issue in the Microsoft dataset as there is always some

other non-hexadecimal token before them, causing them to be excluded in our extraction. We therefore excluded the "db" at the end of any line (which is always lowercase if it is not an actual byte in the code but added by IDA, so no innocent bytes were lost). With both of these issues it is possible we are disassembling the data differently than Microsoft did in the Kaggle challenge. Because the exact way that the Microsoft dataset was generated is unknown, we present results on both datasets with the assumption that our benign files are not, but may be, biased."

After addressing the aforementioned issues we mixed benign and malicious samples for a resulting dataset of 9,100 files, split into 7,500 and 1,600 training data points. The dataset was created by randomly drawing from the malicious and benign files. Again, because of the 50,000 line cutoff, a large number of benign files were unable to be used. A final dataset was created of evenly mixed malicious and benign files for a total of 4,000 testing and 1,000 benign samples.

3.1.4 Dataset Reference Table

We have mentioned many different datasets for many different purposes. To avoid confusion, we will list all datasets in Table 3.1 and label them so they may be referred to later on.

Label	Dataset Source	Line Pad	Line Limit	Train Size	Test Size
1	Dasmalwerk/Windows	Yes	No	160	80
2	Kaggle	Yes	No	400	100
3	Kaggle	Yes	Yes	6,500	2,056
4	Kaggle/Windows	Yes	Yes	5,000	1,200
5	Kaggle	No	No	8,000	2,375
6	Kaggle/Windows	No	No	7,500	1,600

Table 3.1: Dataset Summary and Labels

In addition to these main datasets, there are other datasets that will be referred to later on for a specific purpose. These are mostly based on Dataset 5, the Microsoft Malware

Classification Challenge dataset without line padding or a line limit, and are identical in all ways mentioned in Table 3.1. We will define them in Table 3.2.

Label	Alteration
7	No Random Pooling Used
8	Frames Overlap by 17 lines
9	No Embedding Used
10	0-Day Malware (based on dataset 6)

Table 3.2: Other Datasets

Most of the datasets in 3.2 are self-explanatory, however, dataset 10 requires more detail. Dataset 10 was based on dataset 6 but with a twist: malware specifically of class 8 was excluded from the malicious set. In this set, malicious vs. benign samples are compared. To more thoroughly test the network we explore 0-day malware, or malware never seen before. When making the dataset, class 8 malware was excluded. Then, after the network was trained on this dataset, class 8 malware samples were presented, simulating never-before-seen malware. Class 8 was chosen arbitrarily.

Figure 3.5 shows a box-and-whisker plot without outliers for both the Microsoft Malware Classification challenge dataset and the Windows dataset that we created. The Windows dataset is classified under class 10. These plots represent the number of lines in the .text section of the executables, for files that had an extractable .text section.

Figure 3.5 shows that the length of a file can vary wildly per class. Figure 3.6 shows a version of the plot for the classes that are smaller.

From Figures 3.5 and 3.6, we notice that the Windows dataset files are, on average, much longer than any malicious class. There is significant overlap, however, so it is unlikely that a file could be classified based on length alone.

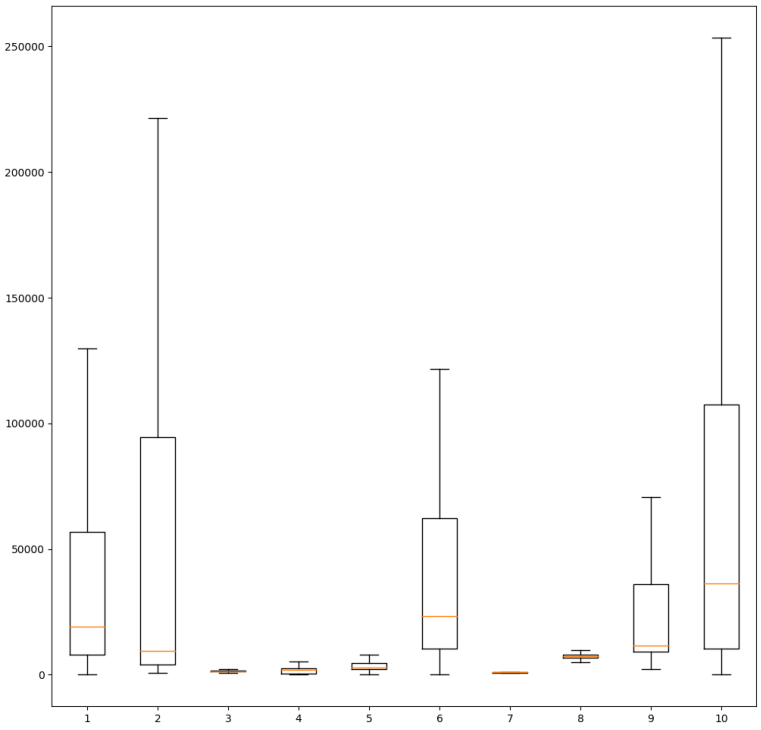

Figure 3.5: Boxplot for All Classes

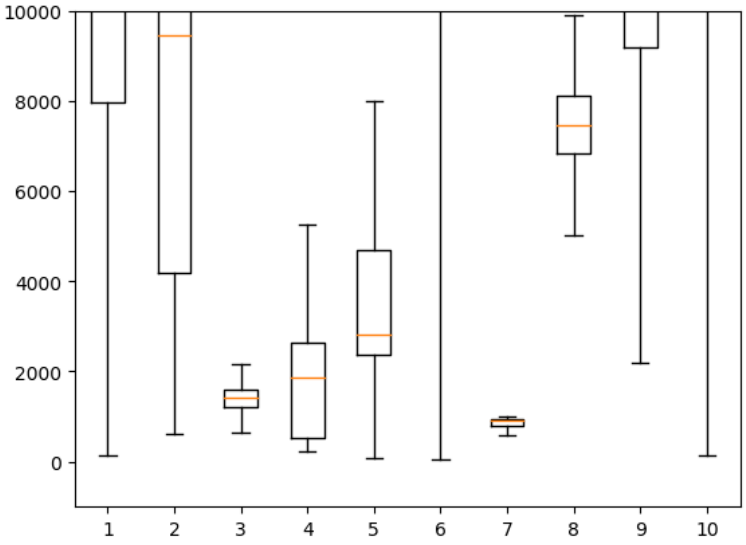

Figure 3.6: Figure 3.5 for Smaller Classes

3.2 Network Architectures

Neural networks have only one constant in seminal literature: that they are in flux. With the given input described, especially MaV, there are infinite neural network architectures that could be attempted for classification. Here we will go through the progression of the neural networks used as each network attempted gave rise to some insight that lead to the next, leading to the final, time-distributed architecture. It is by no means certain that this architecture is the final architecture for all time but it is the overall best we found considering what we learned as the project progressed.

Perhaps the most surprising aspect we have found is that there is seemingly minimal or no temporal dependency between windows or even individual lines of code when it comes to classification. Intuitively, code and programs are vitally dependent on state of the code and past results. We began by assuming that this dependency would extend to a a neural network in that a recurrent network would perform best due to its ability to make judgments based on past information. As will be discussed, however, we found that this is not true.

3.2.1 Common Network Elements

There are some elements that we use throughout all or nearly all neural network architectures shown, namely, the beginning and ending of the networks.

The first layer of almost all neural networks is an embedding. Some other works we have looked at using embeddings [50], which are a popular technique for framing unrelated input data in a more meaningful context. They have proven especially useful in natural language processing [54]. In our case, the value of any single byte does not have meaning in itself. Unlike in a regular image or video where the RGB or gresycale value of a pixel has inherent meaning, i.e. the intensity of a pixel or a truck being blue instead of green, the values of the pixels in bytecode images and videos do not mean anything to an observer. For example, the decimal value for “mov” may be 25, the value for “add” may be 100 and the

value for “eax” may be 26, but that does not mean that “mov” is any closer or more related to “eax” than “add”. Similarly, in natural language processing, assigning numbers to words leads to situations where some words end up closer in value but not necessarily meaning. The purpose of an embedding, then, is to be a mapping to find related tokens and can represent this relation by blowing one dimension into a specified number of dimensions. With more dimensions comes more ways for two points to be closer or further apart. The embedding layer in Keras is a randomly initialized matrix that functions by using each input token as the index. The index corresponds to one unique row in the matrix, which then replaces the token. This look-up-table (LUT) itself does not have a gradient like a neuron does, however, Tensorflow (or other automatic differentiation engines) are able to optimize the outputs for each token. We found that using an embedding improves the results of the network so significantly that it is absolutely necessary, especially in regards to testing accuracy.

After some arbitrary neural network we generally use global average pooling to reduce the number of dimensions of the output before using a densely-connected layer for the final output. Depending on the architecture, the output of the network may be four or five dimensions, so global average pooling can reduce that down to one dimension before the dense layer. Global average pooling has been shown to improve generalization, or testing accuracy, of a neural network [51]. In fact, the alternative to global average pooling would be to use another aggregator like global max pooling or to flatten the output into one dimension. We find that using either of these alternatives would result in a strictly worse output or even result in a network that is extremely fragile in training.

3.2.2 Convolutional Network

Our first priority was to create a network that was similar to other works as a proof-of-concept. The network created in [43] was therefore a standard convolutional neural network as shown in Figure 3.7.

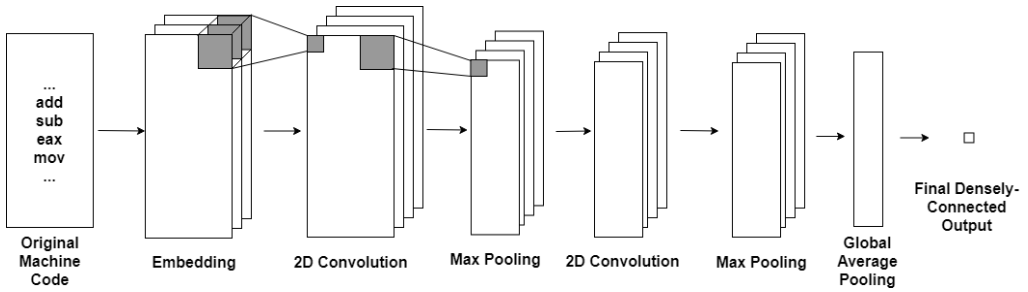

Figure 3.7: Convolutional Network

The network included two sets of a convolutional layer followed by maxpooling, a standard practice for convolutional neural networks. We find that using two convolutional layers, not more or less, was important for the performance of the network. All values for the size of the network were found experimentally. The network was run on two datasets, Dataset 1 and Dataset 3. The parameters for the network for Dataset 1 are shown in Table 3.3, trained with rmsprop and categorical crossentropy.

Layer	Type	Number of Neurons	Activation	Notes
1	Embedding	100	N.A.	Vocab size 256
2	Convolution	400	tanh	Kernel Size 4
3	Max Pool	N.A.	N.A.	Kernel Size 3
4	Convolution	300	tanh	Kernel Size 3
5	Max Pool	N.A.	N.A.	Kernel Size 3
6	Global Average Pool	N.A.	N.A.	N.A.
7	Dense	9	softmax	Output

Table 3.3: Convolutional Network Parameters for Dataset 1

The parameters for the network for Dataset 3 are shown in table 3.4, trained with rmsprop and binary crossentropy.

As will be shown in the results section, these networks left much to be desired when it came to performance. It was a safe assumption that the cause was largely the amount padding used. These datasets limited the input to 50,000 lines, however, the minimum number of lines is less than 250 and the dataset is heavily biased towards smaller files. The next step was therefore to find some way to eliminate the amount of padding without using preprocessing, leading to MaV as previously presented in Figure 3.3.

Layer	Type	Number of Neurons	Activation	Notes
1	Embedding	100	N.A.	Vocab size 903
2	Convolution	200	tanh	Kernel Size 3
3	Max Pool	N.A.	N.A.	Kernel Size 3
4	Convolution	200	tanh	Kernel Size 3
5	Max Pool	N.A.	N.A.	Kernel Size 3
6	Global Average Pool	N.A.	N.A.	N.A.
7	Dense	1	sigmoid	Output

Table 3.4: Convolutional Network Parameters for Dataset 3

While these networks did not perform exceptionally it was significant that they were able to make classifications at all. From this basic architecture we learned a few things: that using only the assembly instructions may be a viable way to classify programs, that an embedding greatly helps with performance, and that global average pooling at the end of the network also greatly helps with performance. The remainder of our architecture search essentially boils down to determining the best network to put between the embedding and the global average pooling.

3.2.3 Recurrent, Convolutional Networks

Dividing up the code images into a video is straightforward, however, classifying them can be tricky. The natural next choice for the task is the ConvLSTM from [15], which uses convolution in tandem with a recurrent neural network architecture to capture both temporal and spatial relationships in data. Figure 3.8 shows the architecture of a full ConvLSTM network for MaV classification for any arbitrary hidden state.

Figure 3.8 shows two hidden, recurrent layers. When a recurrent network in Keras has the parameter return_sequences set to true, it will return the output at each timestep instead of just the final timestep [11]. Using this parameters allows the network to project the feature space onto a new space throughout time and not just before and after, and enables stacking multiple recurrent layers. We found that using two layers, as in the vanilla convolutional neural network, obtained the best performance and experienced the least

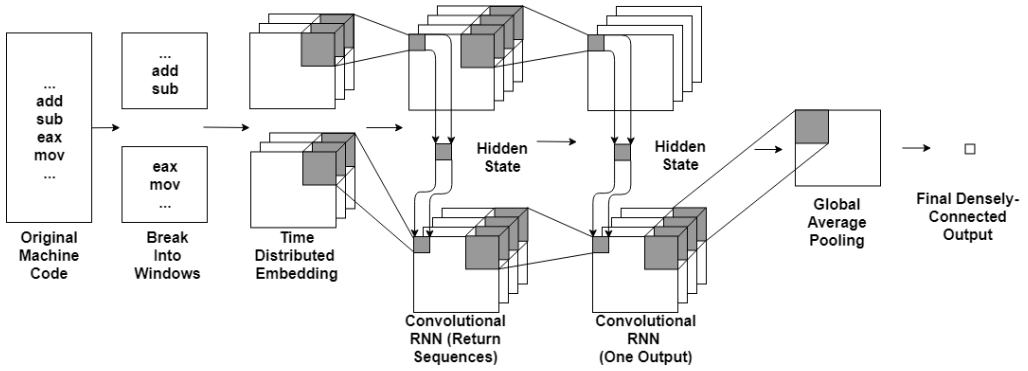

Figure 3.8: Convolutional, Recurrent Network (from [44])

problems in training.

The ConvLSTM network was trained on Dataset 3 and Dataset 4, the datasets that still use line padding and a line limit. All values for the size of the network were found experimentally. The parameters for the network for Dataset 3 are shown in Table 3.5, trained with rmsprop and categorical crossentropy.

Layer	Type	Number of Neurons	Activation	Notes
1	Embedding	100	N.A.	Vocab size 903
2	Lambda	N.A.	N.A.	For Reshaping
3	ConvLSTM	40	sigmoid	Kernel Size 4
4	ConvLSTM	25	sigmoid	Kernel Size 3
5	Global Average Pool	N.A.	N.A.	N.A.
6	Dense	9	softmax	Output

Table 3.5: ConvLSTM Network Parameters for Dataset 3

Layer 2 of the ConvLSTM network is a *Lambda* layer, a custom layer in the Keras framework [11]. The Lambda layer can be written to any purpose. Here, the Lambda layer is what actually reshapes the images into a video. This layer could be implemented in many ways in practice, directly or indirectly, and the embedding does not need to come before it.

The parameters for the network for Dataset 5 were the same as that in 3.5, except that the last layer was only one neuron with a sigmoid activation.

While the ConvLSTM network was working well, it took a large amount of time to make a prediction. The network ultimately used for malware classification must be as fast

and lean as possible to be usable on a wide array of systems. In an attempt to discover what exactly was working about the network we looked at removing parts of the ConvLSTM in a similar fashion to [14], where they looked at what parts of a LSTM are the most important. For this dataset, however, a surprising conclusion was found: that nearly all of the recurrent parts of the network were non-vital. Through trial and error, part after part were removed until the equations in 3.1 were found.

$$\begin{aligned} h_t &= x_t * W_t + b_t + h_{t-1} \\ y_t &= \sigma(h_t) \end{aligned} \tag{3.1}$$

Where σ represents the sigmoid function, W_i and b_i are trainable weight vectors, h_t is the hidden state, and y_t is the output at time t.

Eq. 3.1 shows a barely recurrent neural network that we call the MinConvRNN, or Minimal Convolutional Recurrent Neural Network. Not meant for widespread use, the network was made solely to investigate the theory that the recurrent part of the ConvLSTM may not be necessary in this domain. Before discussing the MinConvRNN, we will first derive its backpropogation equations for completeness. This section is copied from our work done in [44].

"Backpropogation through a recurrent neural network is done through a widely-known procedure called Backpropogation Through Time or BPTT, for short. The procedure consists of unrolling the one neuron into t neurons and repeatedly applying the chain rule for the final derivatives. For this simple network, backpropogation can be done in just a few lines. We will start by defining all parameters in Table 3.6, for an arbitrary objective function J.

The goal of backpropogation is to find the partial derivative of the objective function with respect to a parameter. In this instance, $\frac{\partial J_t}{\partial W_t}$. Note that W_t for time t is a formality, and $W_i = W_j, \forall (i,j) \in t$, and the same for b_i. We begin by defining the base case of $t = 1$ such that $h_0 = 0$, making $h_1 = x_1 * W_1 + b_1$.

Variable Name	Purpose
x_t	Input at time t
h_t	Hidden state at time t
W_t	Weight vector at time t
b_t	Bias vector at time t
y_t	Output of neuron at time t
σ	Sigmoid Function
δ_t	$\frac{\partial J_t}{\partial y_t}$
J_t	Arbitrary Objective function value at time t

Table 3.6: MinConvRNN Backpropogation Parameters

Then, we can define the partial derivative of J with respect to W_t using the chain rule:

$$\frac{\partial J_t}{\partial W_t} = \frac{\partial J_t}{\partial y_t} \cdot \frac{\partial y_t}{\partial h_t} \cdot \frac{\partial h_t}{\partial W_t} \tag{3.2}$$

We can define δ_t to be $\frac{\partial J_t}{\partial y_t}$ to simplify notation as the expression will change for a different J. We can then calculate the other terms directly as:

$$\begin{aligned} \frac{\partial y_t}{\partial h_t} &= \sigma'(h_t) \cdot 1 \\ \frac{\partial h_t}{\partial W_t} &= \frac{\partial x_t * W_t}{\partial W_t} \end{aligned} \tag{3.3}$$

We leave $\frac{\partial x_t * W_t}{\partial W_t}$ alone to simplify the expression, but we note it is computable. It is also known commonly known that $\sigma'(x) = \frac{\partial \sigma(x)}{\partial x} = \sigma(x) \cdot (1 - \sigma(x))$, and we will use $\sigma'(x)$ for simplification of expressions as well.

The full partial derivative at time t can then be defined as:

$$\frac{\partial J_t}{\partial W_t} = \delta_t \cdot \sigma'(x_t * W_t + b_t + h_{t-1}) \cdot \frac{\partial x_t * W_t}{\partial W_t} \tag{3.4}$$

Because W_t is the same $\forall t$ as stated earlier, we can compute the partial derivative for

W for n timesteps as:

$$\begin{aligned} \frac{\partial J}{\partial W} &= \sum_t \delta_t \cdot \sigma'(x_t * W + b_t + h_{t-1}) \cdot \frac{\partial x_t * W}{\partial W} \\ h_0 &= 0 \\ t &\in [1, 2...n] \end{aligned} \tag{3.5}$$

For a recurrent neural network where only the output is considered, we could factor out δ_t from the sum. We can also perform a similar process to find the partial derivative for b for n timesteps as $\frac{\partial J}{\partial b} = \sum_t \delta_t \cdot \sigma'(x_t * W + b_t + h_{t-1})$, with the same conditions.

This partial derivative does not look too dissimilar to that of a basic recurrent neuron. The neuron is indeed recurrent in the sense that it is necessary to calculate h_{t-1} in order to calculate h_t, and that the gradient must be propogated starting with the initial output in the same manner. Regardless of these equations, however, the neuron as defined in 3.1 does not make intuitive sense. Taking the raw sum after convolution, in theory, might simply lead to an unbounded output in the hidden state that approaches ∞ (or $-\infty$) for all elements as the number of timesteps approaches ∞ , leading to all 1s (or 0s) in the output with the sigmoid activation."

The MinConvRNN can be compared to a basic recurrent neural network where all weights from the hidden state are 1. This distinction effectively prevents the network from learning any weights that relate past outputs to the present and solely works to take the sum across all timesteps. Programs are inherently temporally dependent. In fact, they are so temporally dependent that a program will almost certainly completely crash if one line of code is missing. Using a recurrent neural network to classify programs is therefore a natural choice and using the MinConvRNN seems like it disregards that information.

Despite all of these issues, we found empirical success with the MinConvRNN. In practice, the output of the final timestep was composed of 1s for only around 35% of the matrix regardless of length, hinting that the convolution kernels were trained to pick out extremely specific features in the data. If there was a strong temporal link it could be expected that

the MinConvRNN would significantly underperform. Even with a much lower number of total parameters due to the smaller hidden state, however, the MinConvRNN achieved only a slight performance decrease. We interpret this success to mean that the temporal link between windows of code is not important for classification.

The MinConvRNN network was trained on Dataset 3 and Dataset 5, the datasets that still use line padding and a line limit. MinConvRNN layers were substituted directly for the ConvLSTM layers, and we trained the MinConvRNN with the exact same number of kernels as in 3.5 specifically to test the performance difference. We will present those results in the next chapter.

One issue with the MinConvRNN is that there is an artifact from its recurrence in the network, the return_sequences parameter. With return_sequences, the output every single timestep is returned, meaning that the intermediate outputs returned represent a partial sum of the full output.

Competitive results have been achieved at this point, along with the novel idea of MaV, but ending the story at MinConvRNN does not make sense. Even though it is a simple sum, MinConvRNN is slightly recurrent and brings baggage with it, namely that it takes a significant amount of time to make a prediction due to the inability of recurrent networks to be run in parallel.

3.2.4 Distributed Convolutional Network

The final class of networks attempted are that of Distributed Convolutional networks. Learning from the MinConvRNN but ditching recurrence entirely, these networks can be seen as taking one small, convolutional network and running it on each timestep individually. In other words, the weaknesses of the MinConvRNN due to the returning of intermediate outputs and inability to parallelize have been eliminated. The network architecture is shown in Figure 3.9.

Figure 3.9 shows all of the famililar elements, including two sets of convolutional lay-

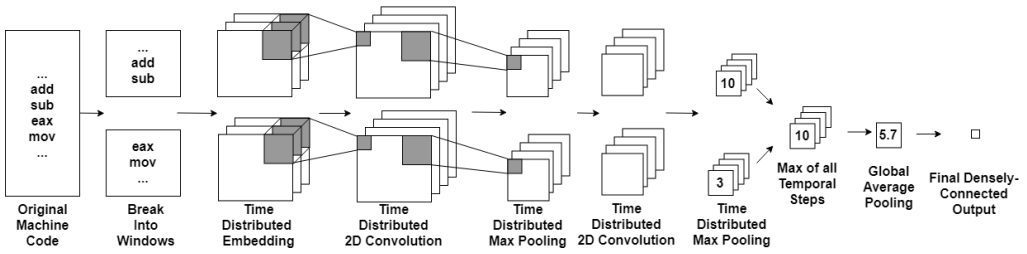

Figure 3.9: Convolutional, Time-Distributed Network (from [45])

ers, except that the convolutional layers are now distributed. Performing this action is simple with the TimeDistributed wrapper in Keras [11]. An important aspect of the distributed network is that it requires some aggregator at the end to turn any number of timesteps into one output, which was previously done by the recurrent network inherently. Here we show Global Max Pooling, which takes the maximum of each element with respect to each timestep. Multiple aggregators were attempted, such as mean, sum, min, or combinations of all of these, but we found max to be the best performing.

Calculating the loss for a TimeDistributed layer, and consequently the gradient, depends on the layers after the TimeDistributed layer. In 3.9, the separate TimeDistributed layers can be considered as one single TimeDistributed mini-network inside the larger network. Gobal Max Pooling functions as the aggregator that comes directly after after his mini-network, meaning the gradient from backpropogation depends on this layer. Similar to local max-pooling, the gradient for Global Max Pooling backpropogates linearly for the winning elements, and is set to 0 for all others. Therefore, all elements for all timesteps have a gradient of 0 except for the winning elements, which are unmodified in the pooling layer.

The network was trained on Datasets 5, 6, 7, 8, and 9, serving as the standard benchmark from here on. Parameters for the network on Dataset 5 were initially based on both the ConvLSTM network and are presented in Table 3.7.

Table 3.7 again shows a Lambda layer for reshaping, and uses a second Lambda layer for the temporal max-pooling. Additionally, another Dense layer was added before the final

Layer	Type	Number of Neurons	Activation	Notes
1	Embedding	100	N.A.	Vocab size 256
2	Lambda	N.A.	N.A.	For Reshaping
3	Convolution	40	tanh	Kernel Size 5
4	Max Pool	N.A.	N.A.	Kernel Size 2
5	Convolution	25	tanh	Kernel Size 3
6	Max Pool	N.A.	N.A.	Kernel Size 2
7	Lambda	N.A.	N.A.	Temporal Maxpool
8	Global Avg. Pool	N.A.	N.A.	N.A.
9	Dense	50	tanh	N.A.
10	Dense	9	softmax	Output

Table 3.7: Distributed Network Parameters for Dataset 5

output for performance. This table comes from our work in [45].

The Distributed Network architecture was a marked improvement over the previous architectures as a culmination of the learnings from them all. As will be discussed, it obtained identical or better performance to the best previous network, the ConvLSTM, while also cutting the amount of time to make a prediction massively.

3.3 Network Compression

Reducing the size and complexity of a malware-detecting neural network is of the utmost importance so that it may be feasible for deployment. In this section, we review the methods of pruning and quantization attempted on the Distributed Convolution network.

3.3.1 Adapting Traditional Pruning Attempts

To prune the network we first attempted to adapt some established pruning techniques to the unique, time-distributed nature of the network. The difficulty with pruning the Distributed Network is that it is not as straightforward as a regular feedforward network. While it is feedforward, a small change in the network can have a massive impact as it will affect all timesteps. Additionally, some methods of pruning use the output of a neuron given some input, and the output of an intermediate, time-distributed layer will be of varying sizes, so it must be aggregated in some fashion. All pruning was done by modifying a copy of the open source library Keras Surgeon [55]. More methods were attempted than those presented here, however, we will only present a couple of the more popular methods that were unsuccessful.

We first attempted to use the minimum weight method by removing kernels with the smallest magnitude [23]. The magnitude can be determined by $C_t(w) = \frac{1}{|w|} \sum_i \sum_j w_{ij}^2$ where *t* is the frame of the input, w_i is a given weight in the kernel, *i* and *j* are the dimensions of the kernel, and $|w|$ is the dimensionality of the weights. We found that this method would destroy network performance even if only one neuron was moved so it was not suitable for our network.

We also attempted a method called Average Percentage of Zeros or APoZ [56], specificalyl targeted for ReLU-activated networks. We built a network with the ReLU activation in one layer to use this method. APoZ specifically removes neurons that activate infrequently and therefore do not contribute largely to the network. Again, using APoZ would

end up rendering the network untrainable, likely due to the unique nature of our network as mentioned earlier. No adaptation was necessary for this method as it would take the average activations of a neuron across all timesteps.

Finally, we attempted a method that attempts to measure the impact of a kernel by using the output vector. Because our network uses max-pooling after performing convolution, kernels that give a low magnitude output can theoretically be removed without issue. We measured the overall impact of a kernel *w* at a frame *t* can be defined as $C_t(w) = \sum_p \sum_n \sum_m |(p * w)(n, m)|$, where *p* is an input vector and *n* and *m* are the dimensions of the output of convolution between the input and *w*. We found this method the most immediately successful by not rendering the network untrainable, however, it was only able to prune a few neurons at most.

3.3.2 Node-Distance Pruning

After experiencing unsatisfactory results with adapted pruning methods we attempted to make a novel pruning method to address the unique nature of the network. Parts of this section, Figure 3.10, and Algorithm 1 come from our work in [45].

The issue with the output magnitude method is that it only looks for the kernels with the largest total output, however, it is not necessarily the largest output that matters, especially considering the first hidden layer of the network.

Instead, it might be better to look at the uniqueness of each kernel. The goal of this method is to build a Node-Distance graph, which measure the distance of each kernel to each other kernel, resulting in a fully-connected graph. Figure 3.10 shows an example Node-Distance graph.

Figure 3.10 shows an example Node-Distance graph consisting of kernels A, B, C, and D, where the edge connecting any 2 nodes represents the distance between those two kernels.

To build the Node-Distance graph we consider the distance between any two kernels

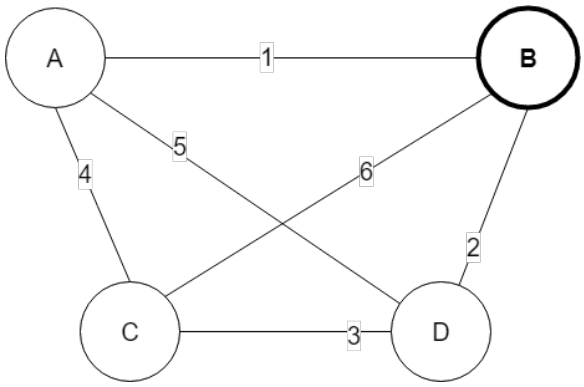

Figure 3.10: Node-Distance Graph for Pruning

k_1 and k_2 to be determined simply by $dist(k_1, k_2, q) = \|(k_1 * q) - (k_2 * q)\|_2$, where q is some input values propagated from previous layers. We use the 2-norm of the difference of the outputs of the two kernels in response to an input. The 2-norm empirically worked the best for the pruning method, however, many distance measures exist, such as cosine similarity. Regardless of how it is measured, the goal of this operation is to find which two kernels are the most similar to each other, in that they end up producing the most similar output.

The neuron to prune can then be determined by taking the lowest magnitude nodes in the graph, determined by summing the edges. The nodes with the lowest magnitude can be considered as the node the closest to many other nodes in the graph, meaning it is the most redundant.

The graph can be built and used as algorithm 1 shows.

To use algorithm 1, each training or testing sample must be run through the network and the output of the desired intermediate layer captured. Then, the outputs are grouped by the kernel that created them and fed to the algorithm.

One note to make of the algorithm is that all magnitudes of all nodes must be recalculated every time a node is marked for pruning (the reason for the outer while loop, and that the distances must be recalculated each loop). This is because when a node is removed, the next smallest magnitude node has the possibility of changing. Figure 3.10 is actually

```
Data: K, a list of all kernel outputs in the network, and C, the number of kernels to
      remove
Result: P, the kernels to remove
Initialization;
P = empty list of kernels to prune
Mag = empty list of len(K) to represent magnitude of each kernel
while len(P) <C do
    for n = 0 to len(K)-1 do
        for m = n+1 to len(K) do
            dist = distance between K[m] and K[n]
            Mag[m] += dist
            Mag[n] += dist
            if m in P then
                Mag[m] = ∞
            end
            if n in P then
                Mag[n] = ∞
            end
        end
    end
    Kernels = Mag.argsort()
    Append Kernels[0] to P
end
```

Algorithm 1: Node-Distance Pruning Algorithm

one such case: node B has the smallest magnitude, followed by either node A or node D. When B is removed, however, node C then becomes the smallest magnitude node. This occurrence is extremely rare in practice, never having happened when we used the pruning algorithm on our networks, but it is possible.

It is quite a slow algorithm. Not counting the time to get all outputs of a network, O(Cn^2) distances will be calculated. We use n = 0 to len(K-1) and m = n+1 to len(K) to reduce the total calculations (seen as calculating only the upper triangular of an adjacency matrix), however it is still O(Cn^2), which is especially large given that calculating the distance between two kernels is non-trivial. A small speed gain could be had by only calculating the distance from each node to each node once, and instead of recalculating all distances every time a node is removed, only subtract the removed distances. Doing this, however, would require storing the full adjacency matrix, where we only store a list of

$len(K)$ elements, and we thought the algorithm may be less clear.

We found this algorithm to be extremely successful on our network by being able to reduce the total number of kernels in the network significantly and never rendering the network untrainable. Specifically, we found that the 3rd layer of the network, or the first distributed convolutional layer, was highly redundant and pruned all the way down to 10 kernels.

The goal of this algorithm can be seen as reducing the variance in the node magnitudes. As the lowest magnitude nodes are removed and the remaining nodes retrained, the magnitudes of the nodes come closer together. We found this to be empirically true. The magnitudes of the nodes in the unpruned network with 30 kernels ranged from 27,546 to 45,172 (the size of these numbers will depend on the number of nodes in the layer and the number of samples given to the algorithm). After pruning, the magnitudes of the nodes in the network with 10 kernels ranged from 10,371 to 12,347, a stark decrease in how much the magnitudes vary. Performing a larger study of this algorithm on more networks would be worthwhile. For instance, if any standardization of the magnitudes helps, perhaps removing the largest magnitude nodes (or most unique nodes) would also be helpful. Other distance metrics could also be considered.

3.3.3 Quantization

This section is an excerpt from our yet unpublished work in [45].

Quantizing neural networks can mean multiple things, but in this context, we use it to mean finding similar weights in neural network layers and replacing them with one shared value. When the network is then deployed in hardware or software, only a limited number of unique values are required. The number of elements is the same but the amount of memory required is highly compressed. Ideally, quantization causes no loss in performance. Making a malware-detecting neural network as small as possible is again highly desired for applications in a commercial environment where the client may not have a powerful

computer, or even an embedded device, such as a phone. The difference between tens or hundreds of thousands of unique weights and hundreds of unique weights may greatly affect real-world applications. We apply a basic quantization method to the network to find the smallest number of unique values required to identify malware.

Work done in [30] compares methodologies for quantizing neural networks including using k-means clustering for creating clusters of weights, and work in [33] uses another clustering method for weight quantization. We similarly make a simple rule of thumb for the number of quantization bins and use a Python implementation of the Fisher-Jenks Natural Breaks algorithm to find the bins, which acts similarly to a 1-D k-means [57]. The number of bins is determined by taking the logarithm of the number of parameters to put in bins. Experimentally, using this rule works well for quantization, and further reduction in the number of bins will decrease accuracy. The algorithm can be described as shown in 2.

Data: All parameters of a given network
Result: Quantized Network
Initialization;
W = a list of each weight, kernel, and bias vector in each layer
for *w in W* **do**
    V = Sort w
    K = log(count(W))
    breakLocs = Jenks(V, K)
    Map each value in V to closest breakLoc
    Replace values in W with map from V
end

Algorithm 2: Quantization Algorithm

Figure 3.11 shows an example of the weights in a layer becoming quantized. Every weight is set to the closest break value found by the Jenks algorithm. Another way to set the new weights might be to set every value between to breaks to be the average of the breaks. Figure 3.11 shows an example of quantized weights, where the horizontal axis is the set of weights, sorted by value. This format is not how distributions are normally viewed but helps show the effect of quantization.

It can be seen in Figure 3.11 that more breakpoints are put at the beginning and end of

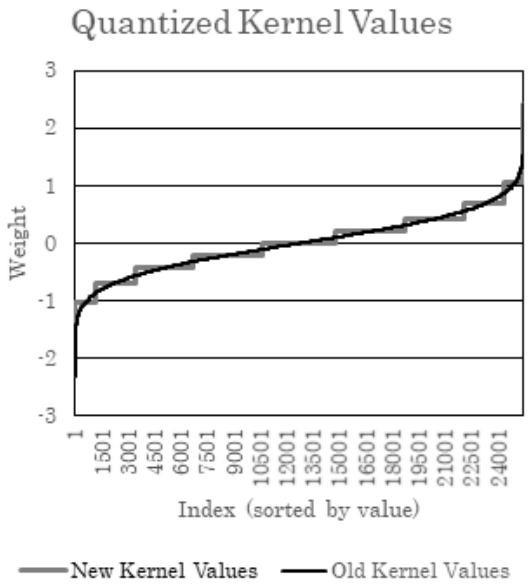

Figure 3.11: Quantized Kernel Values

the set of weights because there is more variation at the beginning and end. Doing this instead of evenly distributed buckets helps the quantization process keep as much information as possible.

We will later discuss the effect of quantization on the performance of the network. One effect to note is that quantization will have more of an impression on the ReLU network than the Tanh network. This is because ReLU does not have an upper bound for the output weight, while Tanh is strictly bounded between -1 and 1. Therefore, changing weights for a Tanh neuron should not affect the performance if the neuron activates to the same inputs. For ReLU, however, changing the weights will not only change if the neuron activates but how much it activates, affecting the performance of the network greatly.

For the quantization it is necessary to decide the number of breaks corresponding to quantization levels q to use. In attempt to make a generalized rule, we use $q = \lfloor log_{10}(N) \rfloor$ where *N* is the number of unique parameters in the layer being quantized. Figure 3.11 shows the set of weights sorted by magnitude. If we view these weights as a line, they take the shape of what looks like one period of a tangent function. In practice, regardless of the

activation function or size of the layer, we find that the set of weights take this shape when sorted distributed across different ranges. Therefore, we determine our quantization rule to be based on log_{10} as exponentially larger layers do not seem to require exponentially more buckets for quantization. It is, of course, easy to incorporate more buckets by using *Nq* buckets, where *N* is some chosen integer.

3.4 Hardware Inferencing Subsystem

The ultimate goal of this work is to have some form of malware-detecting hardware. This section will go through how the network was deployed as a proof of concept to an FPGA. Ultimately, deploying the distributed network is a large undertaking in itself and our goal with this work is to prove that doing so is possible. Many optimizations to make the deployed network work faster as we are only scratching the surface of what could be done.

3.4.1 Full Coprocessing System Overview

Our work has not been done in a vacuum and is part of a larger initiative to make some kind of malware detection coprocessing system. The details of such a system - obtaining the machine code from the processor, decoding the code into assembly, recreating the input program, what to do with an output, and more - is work being done by others in the AFRL research team. Namely, John Musgrave has been investigating how to interface with the CPU and build such a system. Our work is therefore a subsystem for this initiative.

The networks we have developed are a candidate approach for the detection piece of the system. In this case, if the input is determined as malicious, it can be further examined by a model that reasons more in-depth about the code, such as an ACT-R model. We will also discuss using salience as a possible way to explain what warranted a classification in Chapter 5. The overall goal for the system is to make the ultimate decision about a program explainable to a human observer.

Our work is easily extendable to using input from such a system. Using .text from executables is not the exact same but should be analogous enough that a network architecture that works for one input should be able to work for the other. Given that some system exists which hands assembly code to our system, we can design a hardware-based inferencing subsystem to make a classification.

3.4.2 Malware Detection Subsystem Overview

Deploying a neural network for inferencing (or training, which is out of our scope) has become a hot topic in recent times. There are dozens of hardware efforts by companies such as the TPU from Google [58], the Movidius Neural Compute Stick from Intel [59], or the integrated AI processing with SenseMI by AMD [60]. Having hardware that is purpose-built for neural networks enables their use across a wider array of devices and not just systems with top-of-the-line GPUs. We chose to work with an FPGA to keep the network as hardware agnostic as possible, to enable rapid prototyping, and for the open source tools available to ensure we can make the network compatible.

It is important to note that a final malware-detecting subsystem does not have to take the form of an FPGA and could be done with any hardware mentioned before, or could even be done with a GPU as in simulation. Using an FPGA, however, means theoretically not using a full operating system or a CPU - if the prediction was done with a GPU, either a separate virtual machine would need to control the GPU, or a dedicated GPU would need to be installed into a specific PCIe slot on the motherboard for malware classification with CPU support. In other words, some kind of driving CPU would need to control the GPU for loading the model, fetching inputs, and fetching parameters. Regardless of implementation, the method would leave attack vectors open if malware was able to act quickly enough to intercept signals between the CPU and GPU. It might be possible to solder a GPU or other device to a PCB and hard-wire all of the aforementioned functions, however, such an approach would be difficult to design and difficult to adjust if an update of some kind was necessary.

Ideally, using an FPGA means it fully controls itself as it requires no bulky or possibly penetrable operating system. In this case, the primary CPU can directly port assembly commands, in hardware, that it is about to run or is running to the FPGA so that it may check if they are benign. The FPGA could even be integrated directly with the CPU, which has previously shown great promise for malware detection [52].

An extremely generalized image of what our malware-detecting subsystem could look like is shown in 3.12.

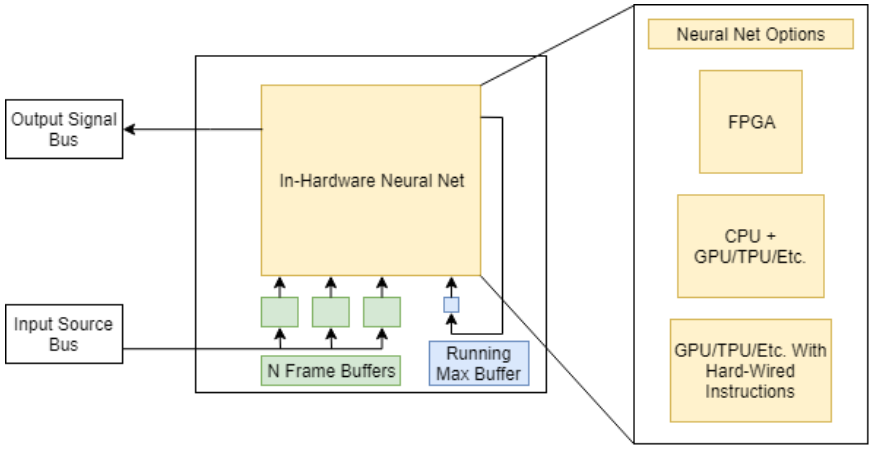

Figure 3.12: Malware Detecting Subsystem

The exact details of the architecture of a malware-detecting subsystem are out of the scope of this thesis, however, Figure 3.12 shows the general idea. The input could even vary depending on the requirements of the system - if fast enough, the system could be a live detection system that uses an input of recreated programs from live commands run by the processor, but if that does not work, the system could function as a gatekeeper that all programs must pass through before executing.

The output bus could also vary. It could be a simple one-bit decision on whether to run something, or the full 32-bit floating-point output, as the final output of the network can be interpreted as the probability a file is malicious. The larger system could then decide to run something in conjunction with other factors, such as how high the probability is, how critical the program is, how critical the system is (a home PC versus industrial safety systems), and so on.

The only requirement for our subsystem would be to have buffers for the input frames and a running max. An arbitrary number of buffers could be used depending on how much room they require, though the more buffers used, the faster the subsystem would be. Each frame can be processed in parallel due to the architecture of the network. The running max

buffer would function as a way to store output for future inputs. If not enough frame buffers were in place for a program, the output of one pass could be stored as a running max in this buffer, for future use (which leads to possibilities for partial file classification, as we will discuss in the Results chapter).

Regardless of the details of the subsystem, our goal here is to evaluate FPGA deployment as an option for the neural network. For our work we used the Terasic DE1-SoC [61] as it was most compatible with the tools we used. This FPGA is not a top-tier model, however, it was the easiest and most straightforward to use. More modern models with more resources on-board would yield better performance.

3.4.3 LeFlow Overview

A multitude of work has also been in progress for neural network acceleration specifically with FPGAs, for example the work done in [62] or more broadly a convolution implementation in [63]. There is also a wide array of synthesis tools to translate various types of code into deployable Verilog [64].

We specifically utilized a tool called LeFlow that takes Tensorflow and converts it into working Verilog [65]. LeFlow not only significantly decreases the workload of translating a neural network to Verilog, but ensures the process is repeatable and dynamic so it may be easily redone if the network ever changes architecture. A commercial, malware-detecting FPGA would be subject to updates as new malware comes out, so using a library like LeFlow would be vital for ensuring that the network could be quickly retrained and deployed. Figure 3.13 shows our interpretation of how simulated neural networks are translated to hardware.

Simulating neural nets, in our case, uses Keras as a means to build the network. Keras uses Tensorflow as a backend (but it can use others), which compiles the code to a combination of C++ and CUDA for use on a CPU or GPU. LeFlow utilizes XLA, a different backend to Tensorflow [12]. Instead of C++ and CUDA, XLA outputs LLVM. Then, a

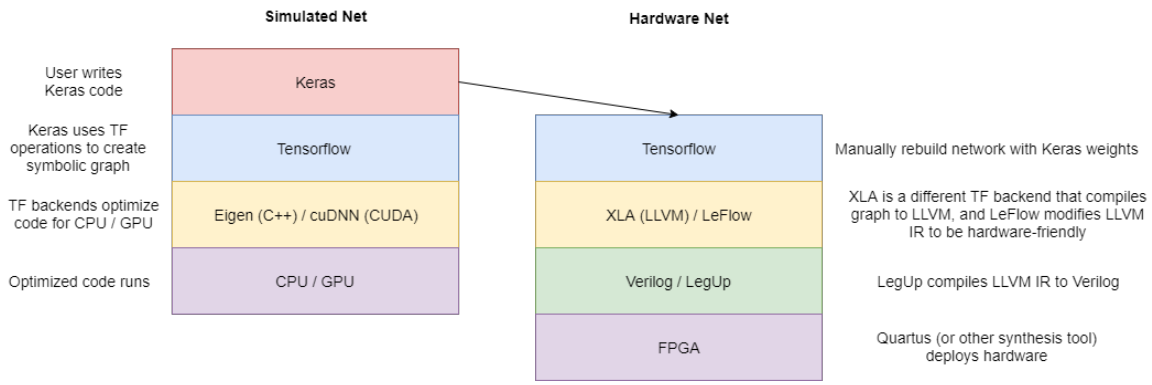

Figure 3.13: Software to Hardware Translation

hardware synthesis called LegUp from the University of Toronto can take the LLVM and compile it to Verilog [66]. We also chose LeFlow due to the open-source nature of the tool. Due to the broad and complex nature of neural networks, it can be easy to find a niche usecase that has not been covered by any tool. Thanks to LeFlow being on GitHub, however, we were able to examine exactly what the tool was doing and make changes where required.

3.4.4 Deploying the Distributed Convolutional Net

The network we are deploying is unique in that it would actually be undesirable to deploy the entire network. More specifically, because the network is time-distributed and repeats the same mini-network over and over, we should instead only deploy that one mini-network, especially because we cannot predict the number of instances needed as the file length changes. We were able to deploy one instance and put one frame of input into memory for classification, however, we were not able to figure out how to pipe an infinite number of frames of input to the network. It would be necessary to write custom Verilog for continuously accepting input from a source like the GPIO or USB. While we could not add this functionality, we can simply extrapolate how long it takes to predict output on one frame of input for predicting how long it would take for the average file.

There were some compatibility issues to address in generating hardware. First, Ten-

sorflow works by building a symbolic graph of operations and thus Keras builds a graph in Tensorflow as well. Due to extra operations that Keras performs, however, it ended up being easier to simply recreate the network in raw Tensorflow rather than using the Keras graph. Recreating the network is not too difficult and doing so allows for greater flexibility in ensuring compatibility, as well as the ability to discard any extraneous parts of the network from Keras.

Issues then came in the compatibility with specific layers and Tensorflow. XLA outputs some LLVM that cannot be interpreted by LegUp, and to solve this, LeFlow uses a custom version of Tensorflow to put the LLVM in the format that LegUp expects. Unfortunately, some more operations needed additional support. Thanks to the tool being open-source and help from its creator, we were able to add support for some of these operations. We added support to LeFlow for handling operations like reduce_mean or or reduce_max, which are used for global average or global max pooling. In this instance, the fix was relatively easy, and only consisted of finding an if-statement in the Tensorflow XLA source to turn off. The if-statement simply turned on or off a process called *vectorizing* of LLVM code, which cannot be handled by LegUp. Therefore, turning it off allowed reduce operations to be synthesized into Verilog. The change was submitted and accepted in a pull request for the LeFlow Github repository.

A larger issue came in the form of the embedding. On the backend, Tensorflow has the ability to use conditionals like if-statements which are required for the Embedding. Unfortunately, conditionals result in *dead nodes*, or nodes that were not taken in the conditional. When these dead nodes are processed through XLA, the output is split into multiple *clusters* and therefore multiple files, which cannot be handled by the toolchain. The end result is that we were unable to use an Embedding for the network. It is likely that using an embedding is possible through modifying the Tensorflow source more extensively.

A quick remedy used was to remove the embedding layer and keep the rest of the network the same. The only changes to the network are to first reshape the input to be in

the form (timesteps, rows, columns, 1), to keep the previous architecture usable, and then to divide the input by 255. Dividing by 255 serves to scale the input between 0 and 1 for easier training. It is important to note that removing the embedding significantly reduced the size and complexity of the network. The full network has around 50,000 parameters or trainable values, while the network without the embedding has around 8,000 parameters. This severe reduction in size contributes to a faster classification time and lighter network however, it also results in a severe decrease in performance, especially for testing accuracy. We present the results of the network without an embedding in the next chapter. The largest concern when it comes to deploying the network on an FPGA is fitting the model into the relatively limited design space and memory. While the model without an embedding was able to deploy to the FPGA it is likely that the full model would not in its raw form. Using the full model with quantization, however, may or may not fit, and further investigation is necessary.

After solving the aforementioned issues, we were able to deploy the network to the FPGA. Deployment was done without the quantization of the weights as custom Verilog would need to be written to accommodate the custom loading of weights. We were also able to use a Phase-Locked-Loop (PLL) to increase the clock speed of the design from the native 50MHz to 100MHz. Instructions for setting up the LeFlow and LegUp environment are in the appendix.

Using the Distributed Network also has a massive untouched area of potential due to the windows not being temporally dependent and using max-pooling as an aggregator. In hardware, the window operations can be fully parallelized. We could therefore deploy as many copies of the distributed network as possible and aggregate the results afterwards.

CHAPTER 4

RESULTS

4.1 Network Architectures

Here we present the results curated from tests on the discussed neural network architectures and eventual hardware deployment. Like in Chapter 3, we will separate results by architecture. To cut down on simulation time and unnecessary figures we will present the results of the Distributed Convolutional network much more in-depth than previous networks as it achieved the best overall performance.

4.1.1 Convolutional Network

The convolutional neural network architecture operated on two datasets, Dataset 1 and Dataset 2. All results are taken from [43] The results of the network on Dataset 1 are shown in table 4.1.

Metric	Convolutional Net
Parameter Count	630,901
Train Accuracy	95.1%
Test Accuracy	94.24%
Test Precision	95.34%
Test Recall	93.18%

Table 4.1: Convolutional Net Results on Dataset 1

The results for the network on Dataset 2 are shown in 4.2.

Metric	Convolutional Net
Parameter Count	1,020,909
Train Accuracy	88%
Test Accuracy	78%

Table 4.2: Convolutional Net Results on Dataset 2

We will not discuss these results in-depth as this network was purely made as a proof of concept and contains certain flaws prohibiting better performance, such as line padding and limiting the number of lines.

Our takeaway from these trials will be one important note; only using the .text section for classification appears possible. At this point, it is not possible to distinguish if the poor performance is due to using the .text section or if it is due to the limitations with the dataset. Future datasets and networks shore up these weaknesses.

4.1.2 Recurrent, Convolutional Networks

In this section we present the performance of the recurrent, convolutional neural networks. All performance is also presented in [44].

Table 4.3 shows performance of the networks on Dataset 3, the padded Kaggle dataset.

Metric	ConvLSTM	MinConvRNN
Parameter Count	573,634	116,951
Train Accuracy	98.35%	97.28%
Test Accuracy	98.24%	96.39%
Time to Inference	∼63.67ms	∼23.73ms

Table 4.3: Recurrent, Convolutional Net Results on Dataset 3

Table 4.4 shows the confusion matrix for the ConvLSTM network on testing data from Dataset 3. The columns are predicted classes and rows are actual classes.

The confusion matrix in Table 4.4 shows a few interesting results. Items per class is biased mostly towards class 3, but are not too heavily biased to the point where it becomes a problem. Class 5 is notorious in the Microsoft Classification Challenge dataset for having an extremely low number of samples and in our testing dataset has a total of 5 samples. The ConvLSTM, and MinConvRNN in 4.5, was unable to correctly classify a single instance of class 5. Another interesting note is that there seems to be a confusion between predicting class 1 for class 8. This issue comes up later on as well, and it is extremely possible to optimize out these confusions, even with something as simple as using weighted categorical

Class	1	2	3	4	5	6	7	8	9
1	249	0	0	0	0	0	0	0	0
2	2	364	0	0	0	0	0	0	0
3	0	0	681	0	0	0	0	0	0
4	0	0	0	99	0	0	0	0	0
5	2	1	0	0	0	0	1	2	0
6	3	0	0	3	0	107	0	0	0
7	0	1	0	0	0	1	89	2	1
8	6	1	0	0	0	1	0	259	1
9	3	1	0	0	0	0	0	4	166

Table 4.4: Confusion Matrix for ConvLSTM2d Net on Testing Data from Dataset 3

crossentropy [67]. The ultimate goal of our research is classifying malicious vs. benign code so no overt optimizations were performed specifically for multiclass classification. We use our performance on the nine classes of malware to strengthen the case for our self-made malicious vs. benign dataset, given that we may have accidentally biased that dataset.

Table 4.5 shows the confusion matrix for the ConvLSTM network on testing data from Dataset 3. The columns are predicted classes and rows are actual classes.

Class	1	2	3	4	5	6	7	8	9
1	245	0	0	0	0	0	0	2	2
2	3	356	1	0	0	2	0	3	1
3	0	0	680	0	0	0	0	1	0
4	0	0	1	98	0	0	0	0	0
5	1	0	0	1	0	1	0	2	1
6	1	0	0	4	0	105	0	2	1
7	0	1	12	0	0	1	80	0	0
8	13	2	0	1	0	0	0	252	0
9	4	1	5	1	0	1	0	2	160

Table 4.5: Confusion Matrix for MinConvRNN on Testing Data from Dataset 3

Table 4.6 shows the performance of the convolutional, recurrent networks on Dataset 4.

Performance from the recurrent, convolutional neural networks is competitive to other methods shown in Background. When it comes to Binary vs. Malicious classification, we

Metric	ConvLSTM	MinConvRNN
Parameter Count	573,634	116,951
Train Accuracy	99.78%	99.32%
Test Accuracy	99.50%	99.20%
Test Recall	99.37%	98.12%
Test Precision	99.58%	99.58%
Time to Predict	∼91.16ms	∼35.32ms

Table 4.6: Recurrent, Convolutional Net Results on Dataset 4

achieve equivalent performance to the best competing paper we reviewed, and for the Microsoft Classification dataset, we achieve competitive performance. We would like to note the possible issues with the Binary vs. Malicious dataset as noted in the Datasets section, however, we see the competitive performance in Table 4.3 to prove that the networks do indeed learn from MaV.

These results are only on line-padded data, and we do not present results of these networks on non-line-padded data for two reasons. First, a significant difference in performance was not found, and second, these results reflect the work performed in [44].

One important note is the comparison in performance between the ConvLSTM and the MinConvRNN. The MinConvRNN was purposely kept to have the same number of kernels per layers in order to keep the comparison as fair as possible. In doing this, the ConvLSTM has a much greater number of overall trainable parameters due to its multiple weight vectors. Even with this advantage, the performance difference is only 2% at absolute maximum in a testing dataset, with a time-to-predict that is twice as fast. These results can be interpreted to show that the strong temporal link in the ConvLSTM is not as important as the actual convolution kernels.

4.1.3 Distributed Convolutional Network

In this section we present the results of the Distributed network. As the best performing network found, it will be tested more thoroughly.

Our first set of results come from a network pruned with adapted, traditional pruning

methods as described in chapter 3. A leaner, better performing network was obtained with the node-distance pruning method though the pruning ultimately was used as an architecture search method. The first set of results is pulled from [45].

We tested the network using both the Tanh activation function for all layers, and ReLU on the first distributed later. We also tested using other activations, such as ELU [68] which should speed up training and improve performance, but found no significant difference. Table 4.7 shows the results of the network on Dataset 5. The first two columns are the final results of networks pruned using adapted traditional methods and the final column is a Tanh-only network pruned with the Node-Distance pruning method.

Metric	Tanh Only	ReLU on Layer 3	Node-Dist Pruned
Parameter Count	124,190		58,544
Train Accuracy	99.83%	99.76%	99.86%
Test Accuracy	98.61%	98.74%	98.74%
log_{10} **Accuracy**	96.17%	94.27%	97.56%
log_{10} **Total Q**	60		57
$4log_{10}$ **Accuracy**	98.48%	98.44%	98.56%
$4log_{10}$ **Total Q**	224		210
Time to Predict	$\sim$2.9ms		$\sim$3.09ms

Table 4.7: Final Distributed Network Results on Dataset 5

In Table 4.7 it can be seen that the distributed network achieved the best performance compared to our other methods, and performance that matches other works we have shown.Furthermore, the network pruned with the Node-Distance method achieved the best overall performance in ever category, which can likely be explained as the network was overfitting ot the training data less. Quantizing the network with $4log_{10}$ levels only resulted in a 0.2-0.3% decrease in performance. Using more quantization levels would decrease degradation further.

Table 4.8 shows the confusion matrix for the Node-Distance pruned, distributed, Tanh-only network on testing data from Dataset 5. The columns are predicted classes and rows are actual classes.

The distributed network performed the best on the Microsoft Malware Classification

Class	1	2	3	4	5	6	7	8	9
1	329	2	0	0	0	0	0	1	0
2	1	536	0	0	0	1	0	3	0
3	0	0	670	0	0	0	0	0	0
4	0	0	0	91	0	0	0	0	0
5	0	0	0	0	3	0	0	1	1
6	0	0	0	2	0	163	0	0	0
7	2	0	0	0	0	0	88	0	1
8	9	0	0	0	0	0	0	251	0
9	0	2	0	0	0	0	0	1	217

Table 4.8: Confusion Matrix for Node-Distance Pruned Net on Testing Data from Dataset 5

Challenge dataset overall. We note that the largest number of errors remains to be confusion between classes 1 and 8, but this time more class 5 is placed correctly.

Table 4.9 shows the performance of the distributed network on Dataset 6, the Benign vs. Malicious dataset without line padding.

Metric	Tanh Only	ReLU on Layer 3	Node-Dist Pruned
Parameter Count	94,841		55,126
Train Accuracy	99.98%	99.99%	99.96%
Test Accuracy	99.36%	99.36%	99.31%
Test Recall	99.27%	99.27%	99.75%
Test Precision	99.51%	99.51%	98.90%
log_{10} **Accuracy**	99.03%	99.20%	99.11%
log_{10} **Total Q**	54		52
Time to Predict	$\sim$8.7ms		$\sim$8.08ms

Table 4.9: Distributed Network Results on Dataset 6

A Receiver Operating Characteristic Curve shows the performance of a binary classification system as the threshold for classification is changed. It would be possible, for instance, to label all files as malware and therefore have 100% accuracy in classifying malware, without being useful. The ROC curve shows the False Positive Rate with respect to the True Positive Rate, so we can see how many false positives the classifier would produce for a given amount of true positives. Figure 4.1 shows the ROC curve, as well as a zoomed in version of the upper left corner.

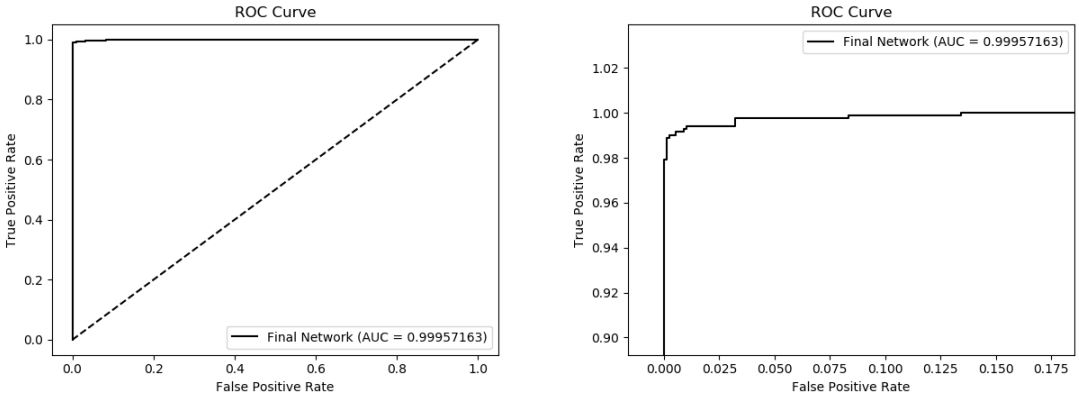

Figure 4.1: Final Binary Classifier ROC Curve on Test Data from Dataset 6

Figure 4.1 shows that the network can identify all malware with a False Positive Rate of 13%, or it can identify 99.89% of malware with a False Positive Rate of 8%, which is competitive with other methods.

K-Fold validation is a common technique used in machine learning to more thoroughly test that the datasets chosen for testing and training are not biased. Theoretically, it is possible to unintentionally choose the testing and training samples such that the network achieves excellent performance. For example, if all of the testing data happens to be the points that are easily distinguishable, perhaps in this case all of the testing points are a specific class of malware that is easily identifiable, it leads to overly confident models.

To combat the possibility of choosing a bad training and testing split, the data is split into k randomly chosen subsets. Then, k times, one subset is used as the testing set while the rest of the data is used for training. A model is then trained and tested on the data and results recorded. An example pseudocode of the algorithm is shown in 3.

To save time, instead of fully training the models we only performed 30 epochs of training. If the models are performing well after 30 epochs we can assume that they could be trained further to represent the results from the previous section. Table 4.10 shows the results of a 10-fold test run, as well as epoch 30 of the network shown in Table 4.7 for comparison.

Data: k, the number of folds, and D, a list of samples
Result: Performance on each fold
Initialization;
foldSize = number of samples per fold, length of D / k
Randomly shuffle D
F = empty list, the list of folds
for *n = 0 to k* **do**
  F.append(D[n*foldSize:(n+1)*foldSize]
end
F now contains a list of k randomly chosen folds
for *n = 0 to k* **do**
  training dataset = combine folds F[:n] and F[n+1:]
  testing dataset = F[n]
  Train an arbitrary model on the testing and training dataset
  Record results for later comparison
end

Algorithm 3: General K-Fold Algorithm

Network	Training Accuracy	Testing Accuracy
Epoch 30 of Fully Trained Network	99.63%	97.89%
Fold 1	99.75%	97.49%
Fold 2	99.71%	96.53%
Fold 3	99.59%	96.62%
Fold 4	99.62%	97.01%
Fold 5	99.80%	97.01%
Fold 6	99.62%	97.78%
Fold 7	99.67%	97.69%
Fold 8	99.59%	95.08%
Fold 9	99.58%	96.72%
Fold 10	99.66%	95.76%

Table 4.10: 10-Fold Validation Performance on Dataset 5 after 30 epochs

Table 4.10 shows that the performance of the various networks on the folds varies between 99.58% and 99.75% for training accuracy, and 95.08% and 97.89% for testing accuracy. Again, the networks could have been trained more. Additionally, these results are from exactly epoch 30, even if a better training or testing performance was obtained an epoch or two earlier.

We see the similar performance across all folds as indicative that the test/training selection we used for training all networks in the previous section was not biased. Differences

between networks is likely the result of statistical noise from randomly initialized weights, floating point roundoffs during gradient calculations, and other various factors.

4.2 Tests Based on Distributed Convolutional Network

In this section we detail more tests based on the Distributed Convolutional Network. We first show a method we call early prediction, which operates by testing the performance of the network on incomplete inputs, strengthening its practical applications. Next, we simulate a 0-day attack, where the network faces a class of malware never seen before. Finally, we present results of training a network on files without an embedding, without pooling, or with window overlap to address claims made earlier in the thesis.

4.2.1 Percent of File

Given the architecture of temporally-independent frames, an interesting question can be posed: what happens if only half of the file is given as input?

If the network is able to classify a significant number of files with only a partial input it would enable faster predictions in practice. There is not reason to not keep track of a running, intermediate output. When the intermediate output indicates malware the program could be then be run with additional restrictions for safety, until it proves itself benign. Other works have struggled specifically with adapting their method to a live detection. These results are from our work in [45].

Figure 4.2 shows the results of running a percentage of every file through the network. Both figures in this section are from [45].

To perform partial file classification, before the file is given to the network, it is multiplied by the percentage on the horizontal axis. If there is not enough lines to make even one frame, it is discarded, resulting in the "jumps" in the graph as large amounts of files can suddenly be used all at once.

To aid in interpreting results, Figure 4.3 shows an expanded view of Figure 4.2.

We see excellent performance of around 90% in classifying as low as 20% of the files. Using 60% of the files results in an accuracy of around 95%.

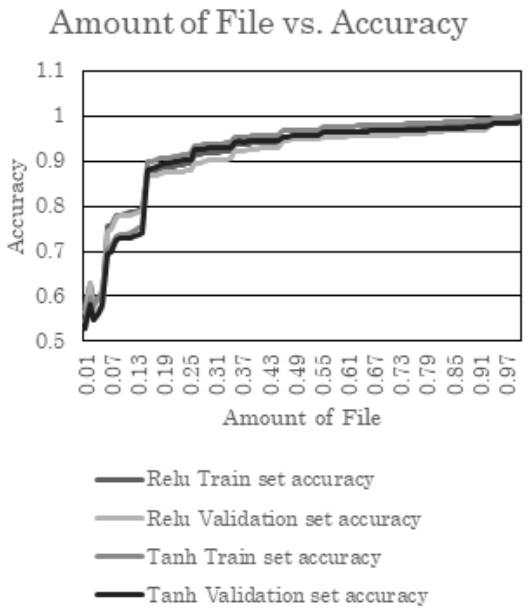

Figure 4.2: Partial File Classification on Dataset 5

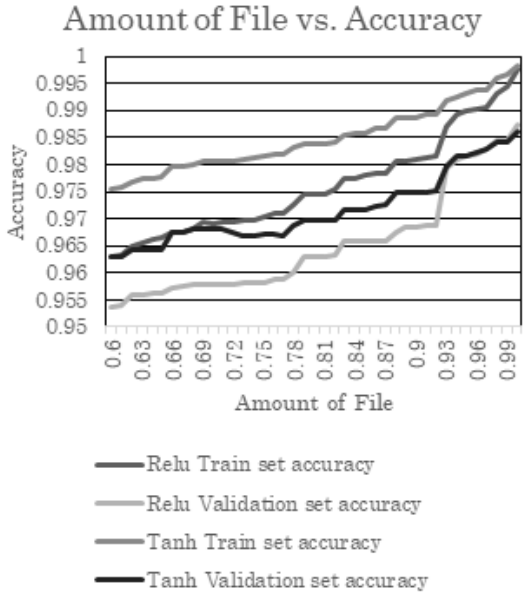

Figure 4.3: Expanded View of Figure 4.2

In practice, if a file is being executed and has been marked as malicious after 60% execution, one could therefore be reasonably confident it is malware.

4.2.2 0-Day Malware

While a network may be able to classify malicious vs. benign files based on files it has seen, there is a major flaw: what if a new type of malware is created, never seen before? This would be called *0-Day* malware and is a large problem for malware detecting systems. These results are from our work in [45].

We used dataset 10 to simulate the appearance of 0-day malware to a network. To do this, we mixed the Microsoft Malware Classification Challenge dataset and the Windows dataset as before, but purposely excluded class 8 files from the malicious dataset. Class 8 was chosen arbitrarily. After training the network on the dataset without class 8, the excluded files were then presented to the network as never-before trained on malware to observe the results. Table 4.11 shows the results of the testing.

Metric	Result
Train Accuracy	99.97%
Test Accuracy	99.45%
0-Day Accuracy	98.28%

Table 4.11: 0-Day Performance on Dataset 10

The network in 4.11 is the same structure as the Node-Dist Pruned network in 4.9, in other words, the best malicious vs. benign dataset. It achieves similar performance to that network though slightly worse, likely due to statistical noise.

The network achieved 98.47% accuracy on 522 files of class 8 malware, a class of malware the network had never seen before. While lower than the full testing accuracy, this performance is highly significant as it suggests one of two things (or some combination of the two):

1. The network is identifying features within assembly code to tell if it is malicious or benign, which can be extendend to 0-day malware

2. The dataset includes some bias, either from the format of the benign code as dis-

cussed, due to the dataset only including Windows files, or due to the malicious code classes all coming from the same source

We have discussed possible issues with the dataset before, however, given that we are controlling for multiple factors with the dataset, we trust that the network achieves competitive performance in identifying 0-day malware for at least this class. It is of course impossible to test for all 0-day malware and more rigorous tests than this can be run, but we see this initial result as highly promising.

4.2.3 Embedding Removal, Data Pooling Removal, and Window Overlap

Here we present some miscellaneous results in order justify earlier claims. Those claims are:

1. Removing the embedding results in a worse but still functional network
2. Using data pooling either does not impact or improves performance
3. Using overlapping frames to more closely resemble a true video does not impact performance

Each of these claims could warrant their own section of tests, however, we only test each claim by training a few networks and observing the best results to not waste time or space. All networks were kept the exact same size as the final Distributed Network size. We used an overlap size of 15 lines for the window overlap dataset.

Table 4.12 shows the results on Dataset 7, 8, and 9.

Metric	Embedding Removal	Data Pooling Removal	Window Overlap
Train Accuracy	97.14%	99.23%	99.85%
Test Accuracy	83.87%	98.23%	98.69%
Time to Predict	∼1.98ms	∼13.14ms	∼4.78ms

Table 4.12: Miscellaneous Results from Final Distributed Network Architecture

Table 4.12 shows that a network without an embedding suffers especially in regards to testing accuracy. The performance drop is unfortunate, however, the network is still useful as a proof-of-concept for hardware deployment. The time to predict did go down with the reduced overall size of the network.

The results for data pooling removal show worse performance in comparison to the results from networks that use pooled data. It is unclear whether the performance hit is within statistical noise or whether the pooling aids in training somehow. Either way, it does not matter, as using data pooling does not decrease performance. More importantly, the time to predict without data pooling increases to six times that without data pooling.

Finally, the results from the window overlapping dataset match results from the dataset without window overlapping. This conclusion matches our theme that the actual order of the windows have little or no temporal dependence. If the dataset had strong dependence, or if extremely specific lines of code were necessary to be analyzed together, performance would change. Instead, we see no performance difference.

4.3 Hardware Deployment

As discussed in the previous chapter we were only able to deploy the network for one frame of input code. The time it takes to make a prediction based on this frame should be the same for any input frame, therefore, we can extrapolate the time it takes to make a prediction on this frame to any number of frames. For our testing we used the cheapest and most compatible FPGA to our toolchain, the Terasic DE1-SoC, an FPGA for educational purposes. This FPGA is the most straightforward to use with LeFlow and LegUp. It is possible to use all of the generated Verilog on a beefier and more modern FPGA, however, it would require integration work to ensure compatibility.

We also used a PLL to increase the clock speed from the native 50MHz to 100MHz. The board can theoretically go up to 500MHz, however, the circuit timing requirements would not allow for a clock higher than 100MHz.

Table 4.13 shows the results of our FPGA testing on one frame of code and extrapolates those results.

Test	Number of Frames	Time to Predict
FPGA Trial with clock at 100MHz	1	∼256ms
Predicted Average per File	87	∼22.3s

Table 4.13: FPGA Performance, Real and Extrapolated

As can be seen in Table 4.13, the extrapolated FPGA classification time unfortunately took about 6,000x longer than the simulated network average of about ∼3.35ms without the embedding (on the malicious vs. benign dataset). While this is a significant decrease, we note that many operations could be vastly improved for reasons discussed before

Figure 4.4 shows the output of the FPGA after running the network.

In Figure 4.4, there are six 7-segment displays. The first display shows the output of the network (in this case, 1, or malicious). The second display shows the output of the global state machine, mostly for debugging and timing purposes. The next four 7-segment

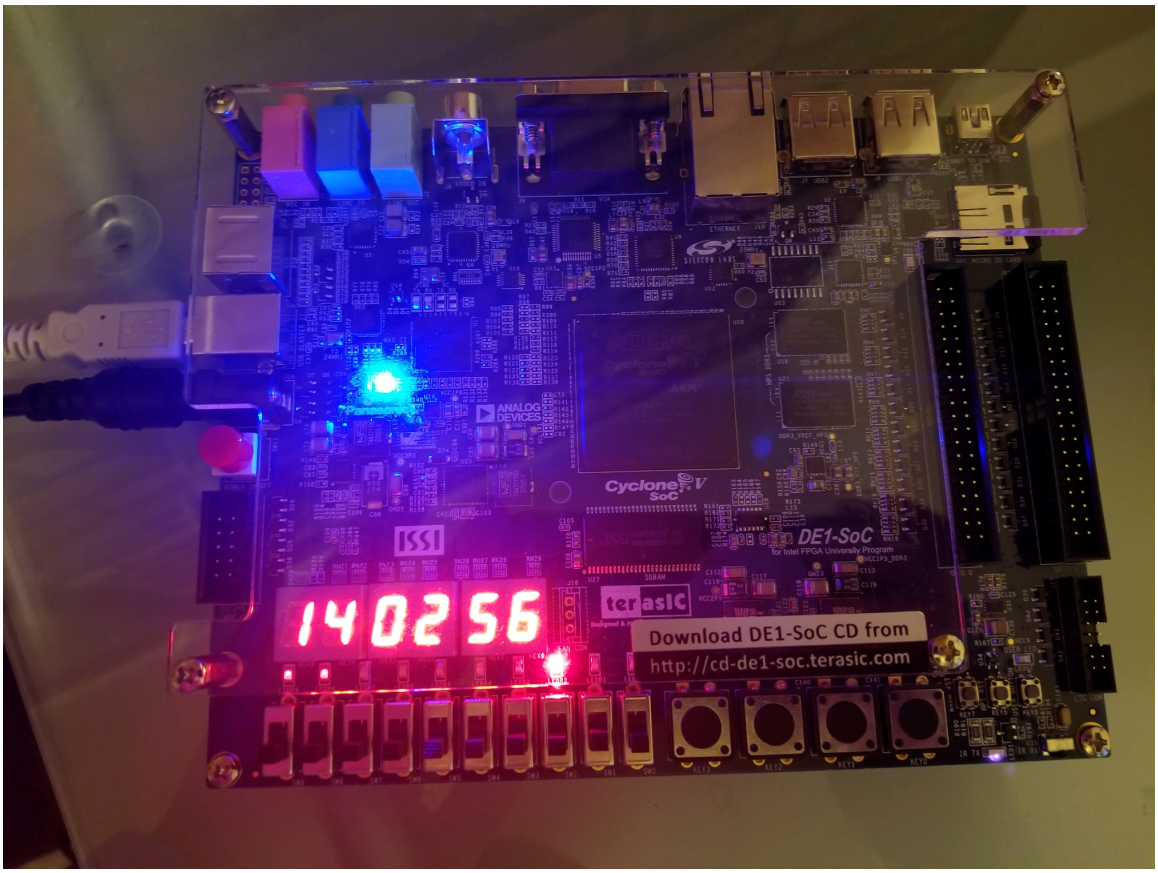

Figure 4.4: FPGA Output After Running

displays show how many miliseconds it took to get the output.

Figure 4.5 shows the output of the Quartus compilation summary.

In Figure 4.5, it can be seen that the FPGA fits the design comfortably. The network uses about 50% of the board's available logic resources and 69% of the board's DSP resources.

The network works quite slowly on the FPGA largely, however, there is a large amount of room for optimizing the generated code. Many other measures could be taken, the most important one being the parallelization of code within modules. Many operations are done suboptimally on the FPGA due to the method of converting synchronous code (LLVM) to asynchronous code (Veriog). LegUp offers substantial support for parallelizing major chunks of the code, however, we did not use any as it would require editing how Tensorflow generates LLVM. Many other optimizations are performed by the authors of

Flow Status	Successful - Sun Mar 3 13:22:20 2019
Quartus II 64-Bit Version	15.0.0 Build 145 04/22/2015 SJ Web Edition
Revision Name	top
Top-level Entity Name	DE1_SoC
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	17,673 / 32,070 (55 %)
Total registers	22217
Total pins	67 / 457 (15 %)
Total virtual pins	0
Total block memory bits	983,934 / 4,065,280 (24 %)
Total DSP Blocks	60 / 87 (69 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	1 / 6 (17 %)
Total DLLs	0 / 4 (0 %)

Figure 4.5: Quartus Compilation Report

[52], who, for example, use a Look Up Table (LUT) to speed up the hyperbolic tangent function. Hyperbolic tangent is expensive to compute so pre-storing values in a table can save large amounts of compute time. Even these simple optimizations could save a large amount of time for the FPGA.

Our initial results are not promising for using an FPGA for classification, however, more work would need to be done to evaluate the usefulness of an FPGA for malware classification. These results prove our main goal in that the network can fit into a low-grade FPGA at all. It is possible that with more optimized code and a modern FPGA the network may approach or exceed the same performance as the GPU performance.

CHAPTER 5
CONCLUSION

5.1 Overview

In this thesis we have investigated the usage of neural networks for malware detection at the assembly instruction level with considerations for hardware deployment. Other works have obtained convincing results that neural networks can classify malware, however, these works have distinct weaknesses in comparison to the work done here. Our main focus is to first limit the input to only the assembly instructions and then rethink the problem from image classification to video classification.

With the imposed limits we then explored possible network architectures for the task, starting with a basic network and evolving that into to a convolutional, recurrent network. Our primary novel contribution is the concept of MaV or Malware-as-Video for classification, as we then found that the temporal component may not be strictly necessary when classifying code. We end with the final, distributed network, which not only operates faster and with better results but also gives multiple freebies like parallel processing or partial file classification. The network was able to achieve convincing performance in comparison to seminal works, largely beating some of the best works we reviewed.

With the distributed architecture we then pruned and quantized the network to better enable hardware acceleration. Another novel contribution is the technique of Node-Distance pruning, which empirically worked well to slim down our network and warrants further investigation. We also then used Jenk's Natural Breaks for quantization. Finally, we experimented with hardware acceleration via FPGA deployment. Results from the FPGA deployment were not stellar, taking multiple orders of magnitude longer to make a prediction than the simulation system. The fact that the network fit on the FPGA at all, however,

is a success and warrants further exploration into hardware-accelerated classification.

Regardless of what specific hardware is used, we have shown that a quick and lean neural network for malware classification is achievable, and using such a network in a commercial system may be viable.

5.2 Network Efficacy and Unknowability

One concern with neural networks across all applications is that they can be hard to decipher. We can discuss the math and see the performance, but ultimately, it can be unclear what exactly the network "sees" in an input to make a classification. This concern is especially important for malware detection as it is vital to ensure the network is working in practice.

Given the high performance margins of neural networks in this area, both in this work and in others, it is safe to assume that there are specific features in code that networks use to make a classification, maybe frequencies of certain opcodes, specific blocks of code that are only used in malware, number and location of branching operations, or maybe an undesirable feature that makes the network not practical.

In this work we make a few observations. Using an embedding is vital for performance, which is not a big surprise. Removing, adding, or changing certain layers hinders performance, which is interesting, but does not answer the larger question. More interestingly, the temporal dependence between windows seems unimportant for classification. This is a unique conclusion, however, it again raises the question of what exactly is important.

These networks may never be used in practice for classifying malware, however, looking into how they work might give vital information for future malware detection methods. One way to look at how networks use input data is to look at the networks *saliency*, which usually entails taking the gradient of the network with respect to its input and feeding it input samples [69]. Performing this operation can give a kind of heatmap for the input, as areas with a large gradient (compared to other areas) indicate an area of sensitivity for the network output. In other words, the network seems to be heavily using that area to make an output decision.

There are many ways to measure saliency and doing so is an area of active research and interest. We took the simplest measure of taking the gradient of the network with

respect to the input and found fascinating results. We can only propogate the gradient to the embedding layer, as it has no gradient, resulting in many gradients, one for each dimension of the embedding. We took the mean, maximum, and minimum of each pixel, to make a few different images, as shown in that order in Figure 5.1.

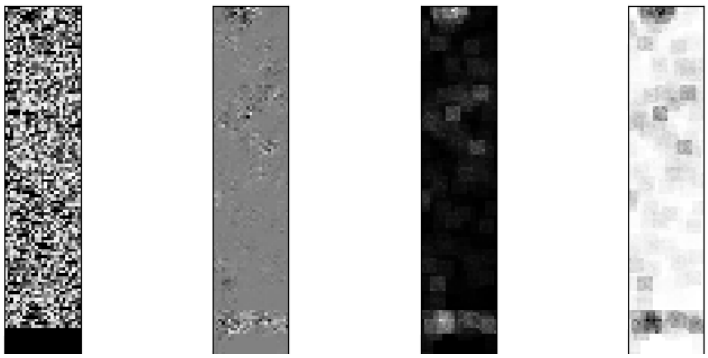

Figure 5.1: Malware Program Heatmaps

Figure 5.1 shows the original malware image, then the mean, maximum, and minimum of the embedding gradient, from left to right. The images have clear points of a large gradient with respect to the input for the network. The gradient in these images is very small. For background areas, they are 0 at numerical precision, and for areas of high activity, they reach only 3-5 magnitudes of order above 0 for numerical precision. All heatmaps from our dataset are identical in nature. Another example shown in Figure 5.2, which shows the heatmaps for a benign file.

These heatmaps are highly though-provoking as they so clearly indicate a small number of high-gradient areas. What happens if we remove areas with a high gradient and replace them with zeroes? What happens if we do so in comparison to doing the same to low gradient areas? What happens if we boost these gradients in testing (can we enable deeper networks)?

Most interestingly, can we map high-gradient areas to sections of code and then make a list of sections common to malware or sections common to benign programs?

Answering all of these questions, and more, would be a substantial amount of work, but

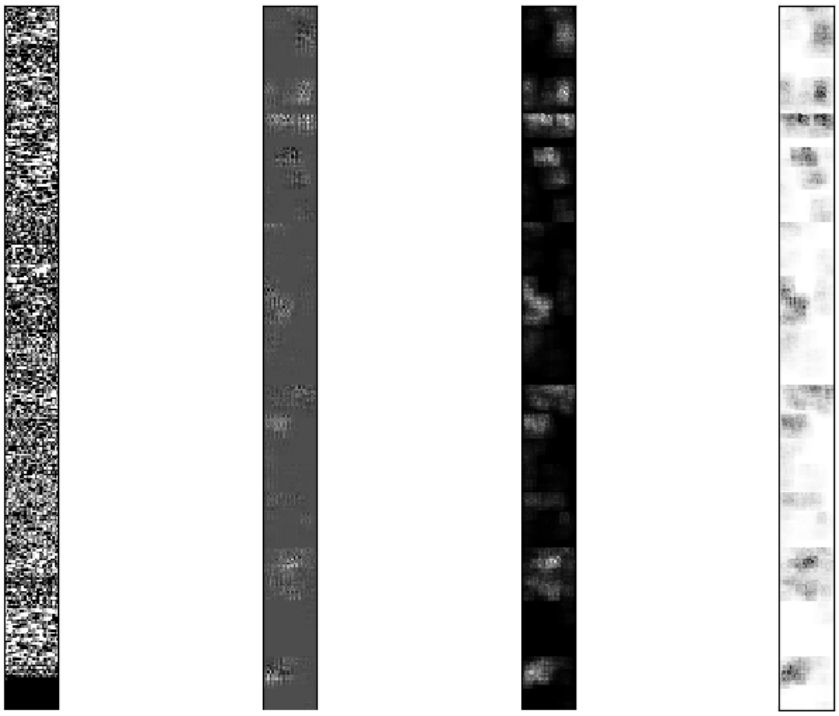

Figure 5.2: Benign Program Heatmaps

could lead to some amazing conclusions about what makes a file malicious. In turn, doing so could lead to even leaner and more robust detection methods. If salience does not pan out for some reason, there are other ways to detect feature importance. Regardless of how it is done, it is important to find why these networks work so well. Ideally, doing so could lead to some realizations about the nature of malware classification.

5.3 Future work

There is a large amount of future work left to be done. Our primary recommendation would be a massive expansion of the dataset. Issues with the benign dataset have been discussed previously, and even within the malicious dataset alone, a more comprehensive dataset with millions of samples would ensure robustness in the network while enabling more rigorous testing. This thesis presents promising results, but ultimately, the testing done does not fully model a real-world scenario given the limited scope of the datasets. One issue with the malicious dataset comes in the form of "injected" malware, or malware that is added to a benign piece of code. We only look at a piece of strictly malicious or strictly benign code, however, malware can hide itself inside what should be a benign file. It would be interesting to test if the network architectures shown could detect that malware as well.

The next area to focus on would be the network itself, given the constantly advancing nature of neural networks. We have seen that the temporal aspect of the code seems to not be important for classification, however, that is not a definitive answer. As discussed in the previous section, using salience or some other means to find more information about the input could lead to leaner and more robust detection methods.

Future work also means ultimately building a fully implemented neural network in hardware and attempting a full system integration. Deploying the network to hardware was the most difficult area for our research due to how deep the field of hardware for neural networks is. It is vital to have a repeatable, easily changeable, and robust system for deployment (in other words, not hand-written Verilog) so that changes to the network do not upend the hardware. Creating and maintaining this system, however, is a large enough amount of work to warrant a in-depth knowledge about neural networks, Tensorflow, Verilog, FPGAs, and so on. Though this effort would not be impossible by any means it would require a substantial amount of work. Whether this system takes the form of FPGA deployment or a GPU/CPU system is up for debate and would require further exploration. Initial

FPGA results are not promising in terms of speed, however, it is exciting that the network was able to fit into the chip.

Given that further testing with new datasets and architecture yields similar success to our own, we hope that the work in this thesis provides a foundations such that a malware-detecting co-processor could become reality.

Appendices

APPENDIX A

SYSTEM SPECIFICATIONS

At points, the time-to-predict is listed for the simulated network, which heavily depends on the system used to run the networks. Table A.1 lists all specifications for the system in which all tests were run.

System Model	ASUS GU501GM Laptop
Processor	Intel Core i7-8750H
GPU	NVIDIA GeForce GTX 1060 (Laptop) - 6144 MB
Memory	16384 MB, DDR4-2666
Python Version	3.6.6
Keras Version	2.2.0
Tensorflow-gpu Version	1.8.0
Conda Version	4.5.12

Table A.1: Simulation System Specifications

The specifications for the FPGA and hardware-synthesizig software used are listed in Table A.2 [61]. The SoC includes a dual FPGA-CPU chip. We only ever use the FPGA, but the CPU specifications are listed.

System Model	Terasic DE1-SoC
FPGA	Cyclone V SoC 5CSEMA5F31C6
Processor	Dual-core ARM Cortex-A9 (HPS)
Logic Elements	85,000
Embedded Memory	4,450 Kbits
FPGA SDRAM	64MB (32Mx16)
CPU SDRAM	1GB (2x256Mx16) DDR3
Quartus Version	15.0
LegUp Version	4.0
LeFlow Commit Used (no releases)	1f1ec0a

Table A.2: FPGA SoC System Specifications

APPENDIX B

HARDWARE SYNTHESIS SETUP INSTRUCTIONS

In this section, we detail the process of synthesizing hardware from a pre-made Tensorflow neural network file (in this case, easy_example.py). We will reference a file, "My First FPGA," which is documentation for synthesizing a design for the DE1-SoC. All of these directions are subject to change per the FPGA used.

1. Download Virtualbox and LegUp VM (http://legup.eecg.utoronto.ca/getstarted.php)
   (a) Give more cores to VM so it runs faster
2. Git pull LeFlow (https://github.com/danielholanda/LeFlow)
   (a) Follow Tensorflow and LeFlow installation instructions (be inside /src/tensorflow)
   (b) Follow these instructions: (https://github.com/danielholanda/LeFlow)
      i. Install Tensorflow
      ii. chmod +x LeFlow (inside /src)
   (c) Also run pip install keras –user
3. Use export PATH="$PATH:/path/to/dir" for LeFlow
   (a) Use /home/legup/LeFlow/src/ as path to dir
4. Create Tensorflow neural net
   (a) Recreate the Keras network purely in Tensorflow
   (b) We have the code for our Keras network in Tensorflow on the Github page
5. At this point, should be able to generate verilog
   (a) From here on, binary_model.py is the file we will use as the neural network in Tensorflow
   (b) Call python binary_model.py to make sure it works
   (c) Call LeFlow binary_model.py
   (d) If you get a bunch of INFO and no ERROR, you are good and binary_model_files/ now exists
   (e) cd into binary_model_files/

(f) Run ”make p”

(g) Call quartus on command line

(h) Follow DE1 SoC documentation My First FPGA from here

i. Open top.qpf

ii. The top-level entity might be the wrong module. If it is ”top”, navigate to Assignments → Settings, and assign it to DE1_SoC. If DE1_SoC does not exist in the project, copy it from our GitHub.

iii. Click on the play button to compile

6. Connecting to board

(a) Adding Drivers

i. Must connect board → Windows → VM

ii. Download drivers from Quartus in Windows

iii. Right click unrecognized device

iv. Add drivers manually

v. https://www.intel.com/content/www/us/en/programmable/downloads/software/prog-software/121.html

(b) In Virtualbox, click on Device → USB → Altera DE1 SoC (NOT Altera - if it just says Altera, board is having trouble connecting)

7. Follow My First FPGA section 4 for deploying to FPGA

APPENDIX C

DE1-SOC TOP-LEVEL ENTITY MODULE

In order to display on the 7-segment displays, we edited the top-level entity to display what we wanted. This module is originally created by LegUp for the DE1-SoC When synthesizing hardware, the top-level entity for the main Verilog file (named the same as the python file) should be set to this entity. If any other modules for other boards are included, they should be deleted.

```
module DE1_SoC (
            CLOCK_50,
            KEY,
            SW,
            HEX0,
            HEX1,
            HEX2,
            HEX3,
            HEX4,
            HEX5,
            LEDR,
                UART_RXD,
                UART_TXD

            );

    input CLOCK_50;
    input [3:0] KEY;
    input [9:0] SW;
    output [6:0] HEX0, HEX1,  HEX2,  HEX3,  HEX4,  HEX5;
    reg [6:0]    hex0, hex1, hex2, hex3, hex4, hex5, hex6, hex7;

    integer clkCount = 0;
    reg clkDone = 0;

    output [7:0] LEDR;
     input UART_RXD;
     output UART_TXD;
         //wire clk = CLOCK_50;
```

```
    wire go = ~KEY[1];

wire clk;
wire rst;
binarypll_0002 binarypll_inst (
            .refclk(CLOCK_50),    //  refclk.clk
            .rst(rst),      //  reset.reset
            .outclk_0(clk), // outclk0.clk
    );



wire          reset = ~KEY[0];
wire          start;
wire [31:0]   return_val;
reg  [31:0]   return_val_reg;
wire          finish;
wire [3:0]    state;

//hex_digits h7( .x(hex7), .hex_LEDs(HEX7));
//hex_digits h6( .x(hex6), .hex_LEDs(HEX6));
hex_digits h5( .x(hex5), .hex_LEDs(HEX5));
hex_digits h4( .x(hex4), .hex_LEDs(HEX4));
hex_digits h3( .x(hex3), .hex_LEDs(HEX3));
hex_digits h2( .x(hex2), .hex_LEDs(HEX2));
hex_digits h1( .x(hex1), .hex_LEDs(HEX1));
hex_digits h0( .x(hex0), .hex_LEDs(HEX0));

    always @ (*) begin
            hex5 <= return_val_reg;
            hex4 <= y_Q;
  //hex3 <= clk;
    end
assign UART_TXD = 1'b0;

parameter s_WAIT = 3'b001, s_START = 3'b010, s_EXE = 3'b011,
            s_DONE = 3'b100;

// state registers
reg [3:0] y_Q, Y_D;

assign LEDR[3:0] = y_Q;

// next state
always @(*)
```

```
    begin
        case (y_Q)
            s_WAIT: if (go) Y_D = s_START; else Y_D = y_Q;

            s_START: Y_D = s_EXE;

            s_EXE: if (!finish) Y_D = s_EXE; else Y_D = s_DONE;

            s_DONE: Y_D = s_DONE;

            default: Y_D = 3'bxxx;
        endcase
    end

    // current state
    always @(posedge clk)
    begin

        if (reset) // synchronous clear
            y_Q <= s_WAIT;
        else
            y_Q <= Y_D;

    end

    always @(posedge clk)
        if (y_Q == s_EXE && (!finish))
            clkCount <= clkCount + 1;
        else if (y_Q == s_EXE && finish && (0 == clkDone)) begin
            return_val_reg <= return_val;
            clkDone <= 1;
            hex3 <= clkCount / 100000000; // seconds
            hex2 <= clkCount / 10000000; // hundreds of ms
            hex1 <=  ((clkCount - hex2 * 10000000) / 1000000);
//tens of ms )
            hex0 <= ((clkCount  - hex2 * 10000000 -  hex1 * 1000000) /
//single digits ms
         end
        else if (y_Q == s_EXE && finish)
            return_val_reg <= return_val;
        else if (y_Q == s_DONE)
            return_val_reg <= return_val_reg;
        else
            return_val_reg <= 0;
```

```
    assign start = (y_Q == s_START);


  top top_inst (
    .clk (clk),
    .reset (reset),
    .finish (finish),
    .return_val (return_val),
      .start (start)

  );

endmodule
```

REFERENCES

[1] D. Gayle, A. Topping, I. Sample, S. Marsh, and V. Dodd, *NHS seeks to recover from global cyber-attack as security concerns resurface — Society — The Guardian*.

[2] N. Idika and A. P. Mathur, "A Survey of Malware Detection Techniques," Tech. Rep., 2007.

[3] Z. Bazrafshan, H. Hashemi, S. M. H. Fard, and A. Hamzeh, "A survey on heuristic malware detection techniques," in *The 5th Conference on Information and Knowledge Technology*, IEEE, 2013, pp. 113–120, ISBN: 978-1-4673-6490-4.

[4] R. Pascanu, J. W. Stokes, H. Sanossian, M. Marinescu, and A. Thomas, "Malware classification with recurrent networks," in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2015, pp. 1916–1920, ISBN: 978-1-4673-6997-8.

[5] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Proceedings - Annual Computer Security Applications Conference, ACSAC*, IEEE, 2007, pp. 421–430, ISBN: 0769530605.

[6] R. Kumar and A. Raj Essar Vaishakh, "Detection of Obfuscation in Java Malware," *Procedia - Procedia Computer Science*, vol. 78, pp. 521–529, 2016.

[7] C. A. Visaggio, G. A. Pagin, and G. Canfora, "An empirical study of metric-based methods to detect obfuscated code," *International Journal of Security & Its Applications*, vol. 7, pp. 59–74, 2013.

[8] Xu Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario, "Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware," in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, IEEE, 2008, pp. 177–186, ISBN: 978-1-4244-2397-2.

[9] S. Ruder, "An overview of gradient descent optimization algorithms," *CoRR*, vol. abs/1609.0, 2016. arXiv: `1609.04747`.

[10] G. Hinton, N. Srivastava, and K. Swersky, "Neural Networks for Machine Learning Lecture 6a Overview of mini--batch gradient descent," Tech. Rep.

[11] F. Chollet *et al.*, *Keras*, `https://keras.io`, 2015.

[12] Google, *XLA — TensorFlow*.

[13] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[14] K. Greff, R. K. Srivastava, J. Koutnik, B. R. Steunebrink, and J. Schmidhuber, "LSTM: A Search Space Odyssey," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, no. 10, pp. 2222–2232, 2017.

[15] X. Shi, Z. Chen, H. Wang, D.-Y. Yeung, W.-K. Wong, W.-C. Woo, and H. Kong Observatory, "Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting," Tech. Rep., 2015. arXiv: 1506.04214v1.

[16] G. Zhu, L. Zhang, P. Shen, and J. Song, "Multimodal Gesture Recognition Using 3-D Convolution and Convolutional LSTM," *IEEE Access*, vol. 5, pp. 4517–4524, 2017.

[17] Z. Yuan, X. Zhou, and T. Yang, "Hetero-ConvLSTM," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining - KDD '18*, New York, New York, USA: ACM Press, 2018, pp. 984–992, ISBN: 9781450355520.

[18] C. Lu, M. Hirsch, and B. Schölkopf, "Flexible Spatio-Temporal Networks for Video Prediction," Tech. Rep., 2017.

[19] J. Donahue, L. A. Hendricks, M. Rohrbach, S. Venugopalan, S. Guadarrama, K. Saenko, and T. Darrell, "Long-Term Recurrent Convolutional Networks for Visual Recognition and Description," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 4, pp. 677–691, 2017.

[20] E. D. Karnin, "A Simple Procedure for Pruning Back-Propagation Trained Neural Networks," *IEEE Transactions on Neural Networks*, vol. 1, no. 2, pp. 239–242, 1990.

[21] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell, "Rethinking the Value of Network Pruning," Tech. Rep., 2018. arXiv: 1810.05270.

[22] E. J. Crowley, J. Turner, A. Storkey, and M. O'Boyle, "Pruning neural networks: is it time to nip it in the bud?" Tech. Rep., 2018. arXiv: 1810.04622.

[23] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning Convolutional Neural Networks for Resource Efficient Inference," *ArXiv e-prints*, 2016. arXiv: 1611.06440.

[24] J.-S. Luo and D. C.-T. Lo, "Binary malware image classification using machine learning with local binary pattern," in *2017 IEEE International Conference on Big Data (Big Data)*, IEEE, 2017, pp. 4664–4667, ISBN: 978-1-5386-2715-0.

[25] R. Reed, "Pruning algorithms-a survey," *IEEE Transactions on Neural Networks*, vol. 4, no. 5, pp. 740–747, 1993.

[26] Q. Huang, K. Zhou, S. You, and U. Neumann, "Learning to Prune Filters in Convolutional Neural Networks," Tech. Rep., 2018. arXiv: 1801.07365.

[27] J.-H. Luo and J. Wu, "An Entropy-based Pruning Method for CNN Compression," *ArXiv e-prints*, 2017. arXiv: arXiv:1706.05791v1.

[28] M. Babaeizadeh, P. Smaragdis, and R. H. Campbell, "NoiseOut: A Simple Way to Prune Neural Networks," *CoRR*, 2016. arXiv: 1611.06211.

[29] J. Proakis and D. Manolakis, *Proakis & Manolakis, Digital Signal Processing, 4th Edition — Pearson*. 2007.

[30] Y. Gong, L. Liu, M. Yang, and L. Bourdev, "Compressing Deep Convolutional Networks Using Vector Quantization," *ArXiv e-prints*, 2014. arXiv: 1412.6115v1.

[31] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both Weights and Connections for Efficient Neural Networks," *ArXiv e-prints*, arXiv: 1506.02626v3.

[32] W. Chen, J. T. Wilson, S. Tyree, K. Q. Weinberger, and Y. Chen, "Compressing Neural Networks with the Hashing Trick," Tech. Rep., 2015. arXiv: 1504.04788v1.

[33] S. Han, H. Mao, and W. J. Dally, "Deep Compression:Compressing Deep Neural Networks with Pruning, Trained Quantization, and Huffman Coding," in *ICLR 2016*. arXiv: arXiv:1510.00149v5.

[34] I. Yoo, "Visualizing Windows Executable Viruses Using Self-Organizing Maps," Tech. Rep., 2004.

[35] J. Z. Kolter and M. A. Maloof, "Learning to Detect and Classify Malicious Executables in the Wild *," *Journal of Machine Learning Research*, vol. 7, pp. 2721–2744, 2006.

[36] L Nataraj, S Karthikeyan, G Jacob, and B. S. Manjunath, "Malware Images: Visualization and Automatic Classification," Tech. Rep., 2010.

[37] A. Makandar and A. Patrot, "Malware analysis and classification using Artificial Neural Network," in *2015 International Conference on Trends in Automation, Communications and Computing Technology (I-TACT-15)*, IEEE, 2015, pp. 1–6, ISBN: 978-1-4673-6667-0.

[38] W. Rawat and Z. Wang, "Deep Convolutional Neural Networks for Image Classification: A Comprehensive Review," *Neural Computation*, vol. 29, no. 9, pp. 2352–2449, 2017.

[39] A. Krizhevsky, I. Sutskever, and G. E. Hinton, *ImageNet Classification with Deep Convolutional Neural Networks*, 2012.

[40] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei, "Large-scale Video Classification with Convolutional Neural Networks," Tech. Rep., 2014.

[41] R. Ronen, M. Radu, C. Feuerstein, E. Yom-Tov, M. Ahmadi, and M. Crowdstrike, "Microsoft Malware Classification Challenge," *ArXiv e-prints*, 2018. arXiv: arXiv:1802.10135v1.

[42] A. Singh, "Malware Classification using Image Representation," Indian Institute of Technology Kanpur, Tech. Rep., 2017.

[43] M. Santacroce, D. Koranek, D. Kapp, A. Ralescu, and R. Jha, "Detecting Malicious Assembly with Deep Learning," in *NAECON 2018 - IEEE National Aerospace and Electronics Conference*, IEEE, 2018, pp. 82–85, ISBN: 978-1-5386-6557-2.

[44] M. Santacroce, D. Koranek, and R. Jha, "Detecting Malicious Assembly using Convolutional, Recurrent Neural Networks," *Submitted but Unpublished*,

[45] M. Santacroce, D. Koranek, and R. Jha, "Exploring the Detection of Malware Code as Videowith Compressed, Time-Distributed CNNs," *Submitted but Unpublished*,

[46] T. M. Kebede, O. Djaneye-Boundjou, B. N. Narayanan, A. Ralescu, and D. Kapp, "Classification of Malware programs using autoencoders based deep learning architecture and its application to the microsoft malware Classification challenge (BIG 2015) dataset," in *2017 IEEE National Aerospace and Electronics Conference (NAECON)*, IEEE, 2017, pp. 70–75, ISBN: 978-1-5386-3200-0.

[47] B. N. Narayanan, O. Djaneye-Boundjou, and T. M. Kebede, "Performance analysis of machine learning and pattern recognition algorithms for Malware classification," in *2016 IEEE National Aerospace and Electronics Conference (NAECON) and Ohio Innovation Summit (OIS)*, IEEE, 2016, pp. 338–342, ISBN: 978-1-5090-3441-3.

[48] T. Messay-Kebede, B. N. Narayanan, and O. Djaneye-Boundjou, "Combination of Traditional and Deep Learning based Architectures to Overcome Class Imbalance and its Application to Malware Classification," in *NAECON 2018 - IEEE National Aerospace and Electronics Conference*, IEEE, 2018, pp. 73–77, ISBN: 978-1-5386-6557-2.

[49] J. Yan, Y. Qi, and Q. Rao, "Detecting Malware with an Ensemble Method Based on Deep Neural Network," *Security and Communication Networks*, vol. 2018, pp. 1–16, 2018.

[50] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. Nicholas, "Malware Detection by Eating a Whole EXE," *ArXiv e-prints*, 2017. arXiv: 1710.09435.

[51] M. Lin, Q. Chen, and S. Yan, "Network In Network," *ArXiv e-prints*, 2013. arXiv: 1312.4400.

[52] M. Ozsoy, K. N. Khasawneh, C. Donovick, I. Gorelik, N. Abu-Ghazaleh, and D. Ponomarev, "Hardware-Based Malware Detection Using Low-Level Architectural Features," *IEEE Transactions on Computers*, vol. 65, no. 11, pp. 3332–3344, 2016.

[53] R. Svensson, *DAS MALWERK*.

[54] T. Schnabel, I. Labutov, D. Mimno, and T. Joachims, "Evaluation methods for unsupervised word embeddings," Tech. Rep., 2015, pp. 17–21.

[55] B. Whetton, *Keras Surgeon: Pruning and other Network Surgery for Trained Keras Models*, 2018.

[56] H. Hu, R. Peng, Y.-W. Tai, and C.-K. Tang, "Network Trimming: A Data-Driven Neuron Pruning Approach towards Efficient Deep Architectures," Tech. Rep., 2016. arXiv: 1607.03250v1.

[57] M. Viry, *Compute Natural Breaks in Python (Fisher-Jenks algorithm)*.

[58] Google, *Using TPUs — TensorFlow*.

[59] Intel, *Intel® Movidius Neural Compute Stick — Intel® Software*.

[60] AMD, *SenseMI Technology — AMD Ryzen — AMD*.

[61] Altera, *DE1-SoC User Manual*, 2015.

[62] M. Rana, D Abdu-Aljabar, and A. Lecturer, "Design and Implementation of Neural Network in FPGA," Tech. Rep. 3, 2012.

[63] H. Ström, "A Parallel FPGA Implementation of Image Convolution," Linköping University, Tech. Rep., 2016.

[64] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A Survey and Evaluation of

FPGA High-Level Synthesis Tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, 2016.

[65] D. H. Noronha, B. Salehpour, and S. J. E. Wilton, "LeFlow: Enabling Flexible FPGA High-Level Synthesis of Tensorflow Deep Neural Networks," Tech. Rep., 2018.

[66] J. Choi, V. Zhang, J. Anderson, A. Canis, M. Aldham, A. Kammoona, S. Brown, and T. Czajkowski, "LegUp: High-level synthesis for FPGA-based processor/accelerator systems Mark Aldham LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems," *ACM/SIGDA 19th International Symposium on Field Programmable Gate Arrays*, 2011.

[67] S. Yue, "Imbalanced Malware Images Classification: a CNN based Approach," Tech. Rep., 2017. arXiv: 1708.08042v1.

[68] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUS)," Tech. Rep. arXiv: 1511.07289v5.

[69] K. Simonyan, A. Vedaldi, and A. Zisserman, "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps," 2013. arXiv: 1312.6034.