Feature Selection for Malware Classification

Mitchell Mays, Noah Drabinsky, Dr. Stefan Brandle
Taylor University

Abstract

In applying machine learning to malware identification, different types of features have proven to be successful. These features have also been tested with different kinds of classification methodologies and have had varying degrees of success. Every time a new machine learning methodology is introduced for classifying malware, there is the potential for increasing the overall quality of malware classification in the field. Even new classifiers with the same accuracy as those used previously can be combined using one of a few different ensemble techniques sharpen the classification and raise the accuracy to new heights.

For our purposes, we have attempted to create a coalition of classifiers which each use different features. These classifiers when trained, provide multiple angles to the same problem and can be used to test ensemble techniques. Eventually, such an ensemble of individual malware classifiers could create a highly precise means of picking out malware from other software.

Specifically, we have created a convolutional neural network which processes byte data as an image, and a deep feed forward neural network which utilizes opcode N-gram features. Both of these classifiers, while not perfect, provide a significant level of classification. They achieve this independently of one another, and when combined, they each contribute enough to improve the final accuracy.

The majority of the effort in this research was placed on gathering the N-gram features, a time and resource intensive process. Tinkering with the parameters or structure of classifiers could provide further improvements to the system.

Introduction

The increased use of technology by citizens in recent decades has made society dependent on it. Technology is not going away anytime soon, and since software has become so entangled in daily life (e.g. monetary transactions, social media, and storage of personal information), it has become an increasingly prevalent target for malicious individuals. With highly motivated and intelligent attackers creating malware, many of the simple defenses currently in place are now insufficient to protect against them. The common anti-virus softwares are unable to guarantee protection against new malware, even if that malware has only been slightly modified from one that is similar.

With this in mind research has turned to machine learning techniques as a possible solution. Anti-virus softwares rely on specific malware signatures (longer byte sequences that are only found in a particular malware executable or highly similar executables). These signatures are a sure sign that a file is a malware. Unfortunately, the trade off of such high confidence is a failure to generalize well to new or unseen malware. Minor changes to a malware can disrupt the signature and allow it to slip past detection. Another method used in current anti-virus software is a collection of heuristics created by experts. These painstakingly created rules attempt to capture the essence of a type of malware. They can do better at generalizing, but the effort needed to keep an anti-virus software up-to-date with new rules for each new type of malware is unsustainable. Machine learning attempts to identify particular indicative features of a malware that can be generalized across all malicious software of its kind.

Previous machine learning attempts have used shorter byte or assembly opcode sequences as a feature. These sequences, called n-grams, are representative of specific actions taken within the execution of a malware. The same actions are replicated in all malware that are in the same family.

For the sake of identifying malware, it is helpful to organize them by family. Each family of malware has a similar course of action or purpose. For example, a Keylogger is a type of malware that records all of the text typed into a keyboard, reporting this information back to the attacker who can extract usernames, passwords, and other important data from it. Although two Keyloggers may accomplish their tasks in unique ways from one another, they both rely on the same essential components to do their work. Therefore, identifying malware by family allows for a system to find multiple types of malware without treating them all as one identical classification. Furthermore, if a particular family of malware can be accurately classified, then the features used to classify it are clear indicators of that specific malware family. Some digging may then uncover how that feature correlates to the malware family in question, providing more insight into how attackers are doing their work.

Related Work

In recent years, the topography of malware research has been churning. Due to a recent surge of success, as well as the nature of the problem, machine learning techniques have served as a backbone for a significant portion of this research. The quantity of effort put forth recently has stemmed from the multiple large scale and damaging malware attacks. Under the public eye, funding and attention have turned to a more innovative solution to the persistent problem. These recent years have brought about many fruitful developments, and have allowed for the production of better protection against malware to be available to businesses and tech-owners.

Schultz, Eskin, Zadok, and Stolfo [1] were the first to publish a work attempting to apply machine learning to malware. They looked at executable files and found features for classification within. Their system used a Naïve Bayes classifier. By training on general indicators from the executable file, they ended up with a system that was significantly more accurate than an anti-virus, although it had potential to falsely identify benign software as malicious.

As with many developments in machine learning, a lot of effort initially was spent on feature discovery. Many components of the executable file were tested as features, including gathering n-grams from the bytes. N-grams were first used in text classification, but Abou-Assaleh, Cercone, Keˇselj, and Sweidan [2] applied this technique for malware classification. They extracted these specific sequences of bytes from the executable files and used their frequency within the file as a feature. The combined power of these n-gram features served as a good classification method for malware.

In more recent years, with a successful set of features for classifying, research has been directed toward different classifier alternatives. This set of possible classifiers including support vector machines, k-nearest neighbor, and artificial neural networks have performed at varying degrees, but many have formed a high accuracy classification system. In the pursuit of the most accurate classifier possible, new research has offered up an ensemble classifier [3, 4]. An ensemble classifier is a classification system which utilizes multiple classifiers to come to a final class decision. The benefit of using multiple classifiers is that where one classifier fails, others may succeed, further cutting down error rates.

Preprocessing

For this research we used a dataset made available by Microsoft through the machine learning website Kaggle. It includes nine different malware families and over 10,000 malwares. Based off of this dataset, we chose to create a classifier to distinguish between the different malware families. The winner of the Kaggle competition provided a paper with the techniques that they used [5]. This served as a launching point for our research. From it, we settled on using a convolutional neural network to treat byte data as an input image, and opcode n-grams for use in a more standard neural network.

Each of the 10,000 malwares provided, had both a byte file and an assembly file, which has been disassembled from the byte file. The byte file was used to create the byte “images” in the following manner. Each byte was converted into a pixel value to be used in the black and white image. Since each line of the byte file is sixteen bytes long, the image created represents this same shape: sixteen pixel wide and sixty-four pixels tall.

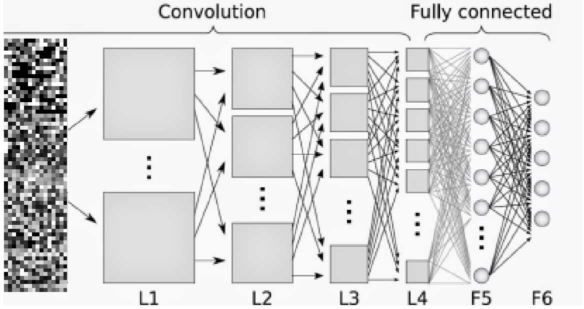

Figure 1. *Example Byte image as input to Convolutional Neural Network*

Figure one shows how a byte image is used as an input to the convolutional neural network. The network performs convolutions on the pixels creating a matrix of smaller convolved images. As it passes through each layer of the network, the information from the image is condensed and routed into a final output layer, giving a value to each possible class node. The class node with the maximum value is the projected class given by the network. Our network had nine class nodes corresponding with the nine possible malware families.

For the second classifier in our coalition, the features were opcode n-grams. One possible n-gram may have been: “mov eax, [ebp+arg_0]”. This is a 3-gram, a sequence of three consecutive data points in the assembly code. For each malware, the number of times that this 3-gram occurred in its assembly would be an input to the classifier. Due to results from Abou-Assaleh, T., N. Cercone, V. Keselj, and R. Sweidan [2] as well as findings from Xiaozhou Wang, Jiwei Liu, and Xueer Chen [5], we settled on n-grams ranging from two to four data points.

The feature gathering for this classifier was more complicated than for the convolutional neural network. A method was needed to identify certain n-grams as possible features and then to decide which of those chosen n-grams would provide the most classification power. There were four stages to this process.

1. 1-gram Selection
2. 1-gram Paring
3. n-gram Selection
4. n-gram Paring

1. 1-gram Selection

Unless intending to evaluate every possible two to four-gram of operations in an assembly file (a number easily in the billions) we needed a criterion to select particular n-grams. One way of doing this is to build larger n-grams off of particular 1-grams. Based off of the work by Xiaozhou Wang, Jiwei Liu, and Xueer Chen [5], we mimicked the execution of the assembly, following every unconditional jump and counting the frequency of each 1-gram. If it occured at least two hundred times in a file it was considered a valid 1-gram.

2. 1-gram Paring

Even after using a specific criterion to limit the number of 1-grams selected, there were too many to build n-grams off of. We used a few further criteria to further pare down the list of 1-grams. First, we removed 1-grams with only one character. We then removed 1-grams that started with a register. While we could assume that the malware would be manipulating the registers, we intuited that having the registers themselves be the start of an n-gram would be unhelpful since it gives no vision into how they are being manipulated. Further paring involved removing items that began with “loc_” or “sub_” as they are somewhat file specific, relying on file locations, and are, therefore, not merely specific to a type of malware.

These heuristic decisions were made due to the limitations of my system being unable to handle the enormous number of n-grams that would be generated from a larger set of 1-grams. It is possible that the 1-grams that were removed could have produced potent n-grams for classification, but based on our reasoning, they were less likely to. After all of paring was complete we were left with 855 different 1-grams.

3. n-gram Selection

With the 1-grams selected, we ran through all of the files a second time, grabbing every possible 2 through 4-gram that began with a 1-gram in the list. Upon completion there were 14,566,780 n-grams. Each with a particular frequency per malware file.

4. n-gram Paring

The most important step to generating an accurate classifier is to select the n-grams from the 14 million that will actually aid in separating some families of malware from others. The best way to evaluate the benefit of an individual feature as an element of a classifier is to test its ability to classify on its own. If a simple classifier trained on the individual feature is able to classify the malwares in a way that exceeds randomness then that feature would be helpful to include in a larger classifier.

Information gain is an algorithm that encapsulates this idea. First, it calculates the entropy in a dataset. Then, it classifies that dataset based on some feature. For each class, the algorithm calculates the entropy of the items within. Subtracting the average entropy of the each designated class from the original entropy gives the information gain, or the improvement in classification ability based off of that feature.

In figure two it is clear that the feature helps to distinguish between the two classes, and the entropy in each new class evidences that. By examining the change in entropy, the helpfulness of a feature is revealed. Unfortunately, to train a classifier on 14 million possible features is a lengthy process. In order to compensate for this I took two different measures.

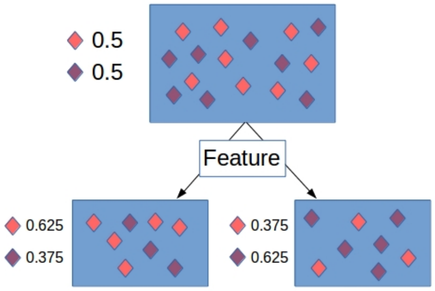

Figure 2. *Change in entropy by classifying on a single feature*

First, rather than running the information gain on each individual feature, we split the list of features into batches of five. Training the mini-classifier on five features instead of one allowed for a quick check to see if any of the five features were helpful. Even with four unhelpful features, we assumed that after training, the one helpful feature would still significantly effect classification, leading to a noticeable information gain.

Second, we distributed the work across several different machines in Taylor University’s Computer Science Department. By re-purposing a bit of code written for distributing graphics rendering tasks, we were able to create a task list of n-grams to run through the algorithm and return the result to a central server. Even with up to sixty-two machines churning through the features, it took three full days to complete the information gain on the 14 million n-grams.

We iterated through the information gain values for each set of five features, finding each set with a value above zero. There were approximately 72,000 sets of five which provided a positive information gain. We split these sets of five into individual tasks and re-ran the information gain test. We were left with 10,985 individual n-grams, to use as features.

Testing

In order to test the the n-gram features, we used a deep feed-forward neural network. This network took 10,985 inputs, one for each n-gram. It had two hidden layers, each with 7,300 nodes. The output had nine nodes, one for each family of malware. The structure of the network was based off of a tutorial by Nathan Lintz [6], and was built using the TensorFlow library.

The convolutional neural network was also built in the TensorFlow library, taking an input of a 16x64 pixel image. It held four convolutional layers and finished with nine output nodes. In order to prevent over-fitting, both networks employed dropout at every layer. Dropout ignores a percentage of the weights in the neural network during training. This prevents the network from hyper fitting to the training data and, therefore, failing on the testing data.

The testing methodology that was used for both of these networks was 10-fold cross validation. The entire set of 10,000 malwares was shuffled and split into ten separate sets with 90% of files used for training and 10% for testing. Each iteration of cross validation took place in the following manner:

First, both networks were trained to completion on the training set. Next, the networks attempted to classify the test sets. The features (byte image or n-gram frequency) were used as inputs to the classifier and the output was a set of float values assigned to the nine class nodes. The node with the highest value was the network's classification. Finally, we utilized the output float values from each classifier to create an ensemble classifier. To get a final classification from the ensemble classifier we multiplied the accuracy of each classifier on the test set by its output values for each malware. We summed these class values for each classifier. This gave a weighted conssensus between the two classifiers.

The feed-forward neural network trained on the malware n-grams twenty times, and the convolutional neural network trained on the 16x64 pixel images fifty times. We chose the

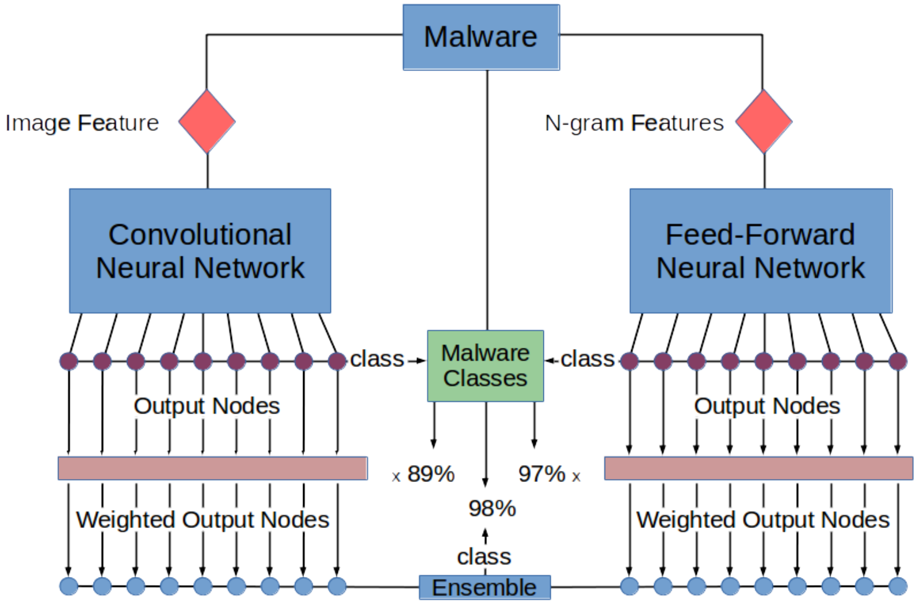

Figure 3. *Testing framework for individual and ensemble classifiers*

number of iterations after overseeing training. Each network took approximately that number of iterations to settle.

Results

After running the cross validations on both of the networks, we were left with a final average accuracy for each. The deep feed-forward neural network using n-gram features trained to be 96.7% accurate. The convolutional neural network, on the other hand, trained to 88.5% accuracy on the test data. By multiplying the nodes by these accuracies, we created weighted nodes. Combining these nodes as an ensemble classifier returned an accuracy of 97.7%.

The training time for the feed-forward neural network was much longer than the training for the convolutional neural network. This was expected since the feed-forward neural network was training weights for every input node rather than the training the smaller convolutions.

Conclusion

The final outcome of my research was an ensemble classifier with 97.7% accuracy on the 10,000 malwares. This value is one percent higher than the best individual classifier in the coalition (96.7% percent from the feed-forward neural network). Yi-Bin, Shu-Chang Din, Chao-Fu Zheng, and Bai-Jian Gao [3] showed that simpler classifiers such as support vector machines and decision trees could be used as components to build an improved ensemble classifier. Our work shows that neural networks can be used as constituents to an ensemble classifier and still create an improved classification.

While limited in our time to tinker with the networks and improve their performance, the principle that ensembles improve the overall classification will most likely still hold as the individual classifiers become more accurate. Furthermore, adding more classifiers to the ensemble should add to the accuracy of the system as a whole.

Future Work

Ultimately, the use of ensembles of complex classifiers has barely scratched the surface of what is possible. Feature gathering took up a large portion of research time, and there is plenty more to be done on the classifiers. Manipulating the structure and number of layers in both the convolutional neural network and feed-forward neural network could drastically effect the performance of the classifiers. Adding new classifiers also has the potential to add significant improvements to the malware classification. Now that using an ensemble method has been applied to neural networks with success, it is worth investing more time to perfect this method and compare it to ensembles which depend on simpler classifiers.

Acknowledgements

[1]. Schultz, M.g., E. Eskin, F. Zadok, and S.j. Stolfo. "Data Mining Methods for Detection of New Malicious Executables." *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001* (n.d.): n. pag. Web.

[2]. Abou-Assaleh, T., N. Cercone, V. Keselj, and R. Sweidan. "N-gram-based Detection of New Malicious Code." *Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.* (n.d.): n. pag. Web.

[3]. Lu, Yi-Bin, Shu-Chang Din, Chao-Fu Zheng, and Bai-Jian Gao. "Using Multi-Feature and Classifier Ensembles to Improve Malware Detection." *Journal of Chung Cheng Institute of Technology* 39.2 (2010): n. pag. Web.

[4]. Kolter, Jeremy Z., and Marcus A. Maloof. "Learning to Detect Malicious Executables in the Wild." *Proceedings of the 2004 ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '04* (2004): n. pag. Web.

[5]. Wang, Xiaozhou, Jiwei Liu, and Xueer Chen. "Microsoft Malware Classification Challenge (BIG 2015): First Place Team: Say No to Overfitting." (2015): n. pag. Web.

[6]. https://github.com/nlintz/TensorFlow-Tutorials/blob/master/03_net.py