Classify Malware using Deep CNN

Ethan Shealey
High Point University Department of Computer Science
High Point University
High Point, North Carolina
eshealey@highpoint.edu

***Abstract*—This experiment is meant to prove the functionality of Depp Convolutional Neural Networks in identifying types of malware.**

I. Introduction

Being able to identify and classify different types of malware has many uses, and could possibly make the internet a safer place. There have already been tools like VirusTotal created to check if a file contains some kinds of harmful code within. VirusTotal uses an extremely large set of antivirus scanners and URL/domain blocklisting services, as well as some other tools. What if, along with those resources, we used a specially crafted Convolutional Neural Network to be able to spot and flag files that might not yet be registered in those databases of known malware.

By converting the binaries of known malware of various kinds into two-dimensional texture images we can visualize the malicious code. Although the produced images are virtually unreadable by the human eye, computers can easily take in the pixels and be able to recognize the patterns occurring that signify malicious code segments, and what kind of malicious code it is. This strategy an be could be utilized to train a Convoluted Neural Network to accurately identify files containing malware, and what kind of malware it is.

II. Malware

Malware, short for ”malicious software”, refers to a software developed by cybercrinimals that are intended to harm a system. This harm can come in many ways like stealing, damaging, or destroying property, files, or hardware. [1] Malware comes in many forms, such as Viruses, Worms, Trojans, Spyware, Adware, Ransomware, and File-less Malware. [1]

Viruses are a malicious software attached to a document or file that supports macros to execite its code and spread from host to host. [1] Once the virus is downloaded, it is capable of laying dormant until the file is opened. They are designed to disrupt a systems ability to operate which can cause significant operational issues and data loss. [1]

Worms are a malicious piece of software designed to rapidly replicate and spread itself on any device(s) within the network. [1] Unlike viruses a worm does not require a host program to spread. A worm begins by getting a victim to download a file, or be connected to a network that someone else has downloaded the file. Worms are capable of causing severe disruptions on all devices it infects. [1]

Trojans are malicious software’s disguised as something useful. Once a victim is tricked into downloading the trojan, it is capable of gaining access to sensitive data and could also modify, delete, or block the data. [1] Trojans are a step up from viruses in terms of harm due to what they can be capable of doing to a victims system.

Spyware is malicious software that is meant to run secretly in the background of a victims system while reporting back to its remote attacker. [1] Rather then blowing its own cover by disrupting the victims system operations, spyware instead sits quietly and collects as much data of what the victim is doing. These kinds of attacks can result in sensitive information being stolen, and could possible grant remote access to the attacker. Spyware is most often used to steal financial or personal information. [1] A common kind of spyware is a key-logger, which logs and tracks all keyboard and other inputs the victim enters. This allows the attacker to view plain text passwords and other critical information.

Adware is malicious software used to collect data on a victims system and provide targeted advertisements to them. Adware is not as dangerous as the others listed, in some cases adware can cause issues for the victims system. [1] Adware will send victims to unsafe, predatory sites intended to infect systems with trojans and other kinds of different malware. Adware software can also eat up a lot of resources and cause a victims system to slow noticeably. [1]

Ransomware is malicious software that gains access to sensitive information within a system, and encrypts it so the victim loses all access to it. The attacker will then demand a financial payout, usually a hefty price, to send the decryption key and allow the victim to recover the lost encrypted data. Ransomware is commonly part of a phishing scam. [1] An attacker would trick a victim into clicking a disguised link, and from there the attack begins. This kind of attack has recently been seen used against hospitals, as most of their systems are heavily outdated and vulnerable. Due to the urgency of the information and operation of their systems the hospitals tend to just pay out the requested amount instead of waiting and/or resetting their systems. These attacks obviously some of the most dangerous forms of malware as they could permanently destroy all information on a person system if they are unwilling or unable to pay.

File-less malware is a type of memory-resident malware. [1] It is malware that operated from a victim's system's memory, not from the hard drive like the others. [1] This results in there being no files to scan for malicious code, making it extremely harder to detect unlike traditional malware. Once the victims system is turned off, the malware disappears, making it also impossible to find during forensics.

III. Convolutional Neural Networks

A Convolutional Neural Network, referred to as a CNN, is a network architecture for deep learning which learns directly from data, eliminating the need for manual feature extraction. [2] Convolutional Neural Networks are particularly useful for finding patterns in images to recognize objects, faces, scenes, and in our case, patterns. They can also be quite effective for classifying non-image data such as audio, time series, and signal data. [2] Using Convolutional Neural Networks for deep learning has become popular due to three main reasons. The first being that Convoluted Neural Networks have no need for manual feature extraction, as the features are learned directly by the Convolutional Neural Network. [2] The second reason being that Convolutional Neural Networks produce highly accurate recognition results, [2] and finally that Convolutional Neural Networks can be retrained for new recognition tasks, meaning you can build on to pre-existing networks. [2]

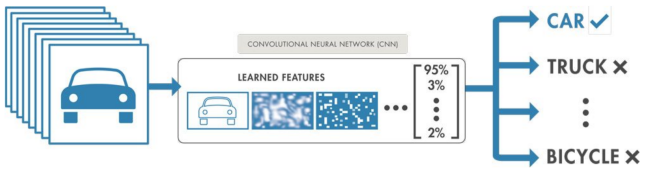

Fig. 1: Deep learning workflow. Images are passed to the Convolutional Neural Network, which automatically learns features and classifies objects [2]

Convolutional Neural Networks provide an optimal architecture for recognizing and learning key features in images and time-series data. [2] Convolutional Neural Networks are a key in technologies like medical imaging, where Convolutional Neural Networks can examine thousands of pathology reports to be able to detect the presence of cancer in images. [2] In audio processing, where Convolutional Neural Networks open the door to keyword detection of specific phrases being spoken, power tools like Siri to be able to recognize phrases needed to do its job. Self driving technologies make use of Convolutional Neural Networks to be able to accurately identify things like stop signs. [2] A Convolutional Neural Network can have tens or hundreds of layers that each learn to detect different features and patterns of an image. Filters are applied to each training image at different resolutions, and the output of each convolved image is used as the input to the next layer. The filters can start as very simple features, such as brightness and edges, and increase in complexity to features that uniquely define the object. [2]

Like other kinds of neural networks, Convolutional Neural Networks are composed of an input, output, and many hidden layers. [2]

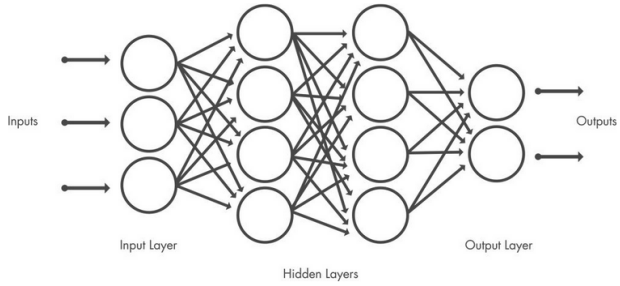

Fig. 2: The different layers of a Convolutional Neural Network. At the beginning there is the input, and in the end the output layers. In between those two layers are a set of hidden layers. These layers perform operations that modify the data with the intent of learning features specific to that data. [2]

There are three different kinds of layers, those being Convolution, Rectified Linear Unit (ReLU), and Pooling. [2] Convolution filters activate certain certain features of the image. [2] Rectified Linear Unit (ReLU) filters allow for faster more effective training by mapping negative values to zero and maintain positive values. [2] Pooling performs nonlinear sampling, simplifying the output and reducing the number of parameters that the network needs to learn. [2] These actions are repeated over over hundreds of layers, with each layer learning to identify different features. [2]

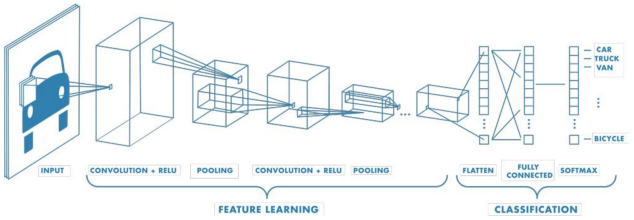

Fig. 3: Example of a network with a couple of convolutional layers. The output of each layer is inputted into the next. [2]

Convolutional Networks tend to be trained with hundreds if not thousands, or even millions of training images. When working with such large sets of data like we are in this experiment, Graphics Processing Units (GPUs) can significantly speed up the time it takes to train the abundance of images. For this experiment we utilized five GPUs to help process the images faster.

IV. Implementation

For implementation, we decided to use MATLAB due to it's built in functionality to make creating and using our

Convolutional Neural Network much easier. Due to the large volume of images we are handling in this experiment, we must use $imageDatastore()$ to load in our data instead of the typical $imread$ as we have done in the past experiments. To do this, we use the following code

$imds = imageDatastore(path, "IncludeSubfolders", ...$
$true, "LabelSource", "foldernames")$

This initializes a new variable called $imds$ that will act as our image data store for this experiment. The parameter $pass$ is the absolute path to the folder containing all of the images we need to train and test our Convolutional Neural Network. $"IncludeSubfolders", true$ enables including all of the subfolders in the specified path given earlier. We must do this for out experiment because out images are located inside subfolders. $"LabelSource", "foldernames"$ makes our datastore use the folder name that the image is located in as that images label.

Since the data that we have read in to our $imds$ variable will act as both our training and testing data sets, we must next split that data into our two sections. We want to do this in a random order, so that every time we run our Convolutional Neural Network, we will not have the same training set thus a better idea of how the system will respond and a better idea of the systems accuracy. To do this we must run

$[train, test] = splitEachLabel(imds, .7, "randomize");$

This utilizes the built-in MATLAB method $splitEachLabel()$. The .7 tells the method we want to split the image datastore 70/30, 70% of the images put aside for training and 30% for testng. With the flag $"randomize"$ included in the method call, it will randomly split the data into the two newly created variables, $train$ and $test$. As the names signify, these will now be our training and testing data sets we will use in our Convolutional Neural Network.

The first problem we will run in to is that our images loaded into our image datastore are not all the same size. To fix this issue we must utilize MATLABs built in $augmentedDatastore()$. This allows us to augment all the images in the original datastore to resize them all into one uniform size.

$auTrain = augmentedImageDatastore([128, 128, 1], train);$
$auTest = augmentedImageDatastore([128, 128, 1], test);$

These lines of code take in our $train$ and $test$ data sets and augment them all to the size $[128, 128, 1]$, and save them into new variables called $auTrain$ and $auTest$. We can now utilize these variables in creating our Convolutional Neural Network.

To create our Convolutional Neural Network, we utilize MATLABs built in $trainNetwork$ method.

$net = trainNetwork(auTrain, layers, options);$

This line will create our Convolutional Neural Network, now called net, using the augmented training data set, $auTrain$. The parameters $layers$ and $options$ tell the method $trainNetwork$ what layers we desire and what options we want to use when creating our Convolutional Neural Network. For this experiment, $layers$ is a matrix consisting of 4 different layers shown below.

$layers = [...$
$imageInputLayer([128, 128, 1]), ...$
$convolution2dLayer(5, 20,' Stride', 1,' Padding',' same'), ...$
$batchNormalizationLayer, reluLayer, ...$
$maxPooling2dLayer(2,' Stride', 2), ...$
$convolution2dLayer(5, 40,' Stride', 2,' Padding',' same'), ...$
$batchNormalizationLayer, reluLayer, ...$
$maxPooling2dLayer(2,' Stride', 2), ...$
$convolution2dLayer(5, 20,' Stride', 2,' Padding',' same'), ...$
$batchNormalizationLayer, reluLayer, ...$
$maxPooling2dLayer(2,' Stride', 2), ...$
$convolution2dLayer(5, 40,' Stride', 2,' Padding',' same'), ...$
$batchNormalizationLayer, reluLayer, ...$
$fullyConnectedLayer(500), fullyConnectedLayer(9), ...$
$softmaxLayer, classificationLayer...$
];

This creates the layers needed for our system to operate properly, consisting of 4 convolutional 2D layers, with a filter size of 5, and the number of filters being 20, 40, 20, and 40 respectively. We also specify the stride as 2, meaning the filter will step 2 at a time, and set the padding to same. In the beginning of the layer matrix, we specify the $imageInputLayer$. This is our entry point into the neural network and we specify the size of the expected inputs to be the same as we augmented the images. Between each layer we include $batchNormalizationLayer$ and a $reluLayer$, and for the first three filters we include a $maxPooling2dLaye$, which we give a pool size of 2 and a stride of 2. At the end of out layer matrix we include a $fullyConnectedLayer$ with a output size of 500, and another with an output size of 9, a $softmaxLayer$, and a $classificationLayer$ which computes the cross-entropy loss for classification and weighted classification tasks with mutually exclusive classes.

The next parameter for the $trainNetwork$ method is $options$. For this experiment these are the options we used.

$options = trainingOptions(...$
$\quad 'sgdm', ...$
$\quad 'MaxEpochs', 40, ...$
$\quad 'ValidationData', auTest, ...$
$\quad 'Verbose', true, ...$
$\quad 'Plots','none'...$
);

For our expirment, we used $sgdm$ as our solver, which uses the stochastic gradient descent with momentum (SGDM) optimizer. We are going to set our max epochs to 40, and use our $auTest$ variable, the augmented testing data set, as our validation data. We wanted our system to be verbose so we can see what is going on behind the scenes to tell what could be going wrong if an issue occurs, and we did not want to plot anything, so we specified that in the options.

Now with these layers and options specified, we can make use of the net variable. To utlizie our new Convolutional Neural Network we have created, we can use the MATLAB $classify$ method.

$guess = classify(net, auTest);$

This segment of code will use our trained Convolutional Neural Network against our augmented testing data set. The results are then stored in the $guess$ matrix. Now that we have the results of the Convolutional Neural Network we created, we can now calculate the accuracy of the system.

$accuracy = sum(guess == test.Labels)/numel(test.Labels);$

This line will take the $guess$ matrix of the answers the system gave us and compare them to the correct test labels from the non-augmented testing data set $test$. Once we have the summed value of all the results we simply divide it with the total size of the $test$ matrix to get the accuracy percentile.

V. Experiment results

For this experiment we used 781 images as our data set for training and testing, stored in folders named one through nine. These folder names will also be our label names. These images consist of thousands of pixels where each color represent code from the binary of the malware. To the human eye these images seem random and we cannot see any patterns, but a computer will be able to form those relations that we are unable to see.

As seen in figures 4 and 5, clearly these images mean nothing to the human eye but hopefully our system will be able to successfully interpret and identify the patterns and features hidden within these large images.

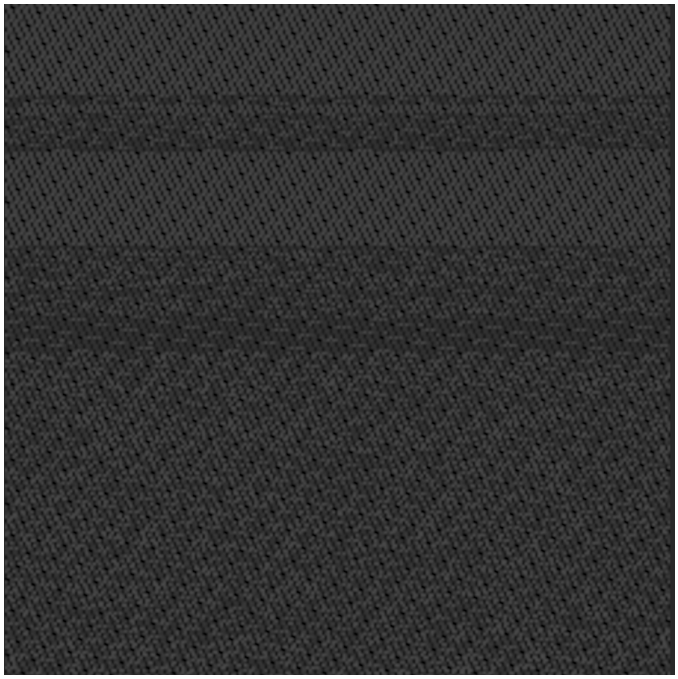

Fig. 4: First example of the images of malware taken from label 1.

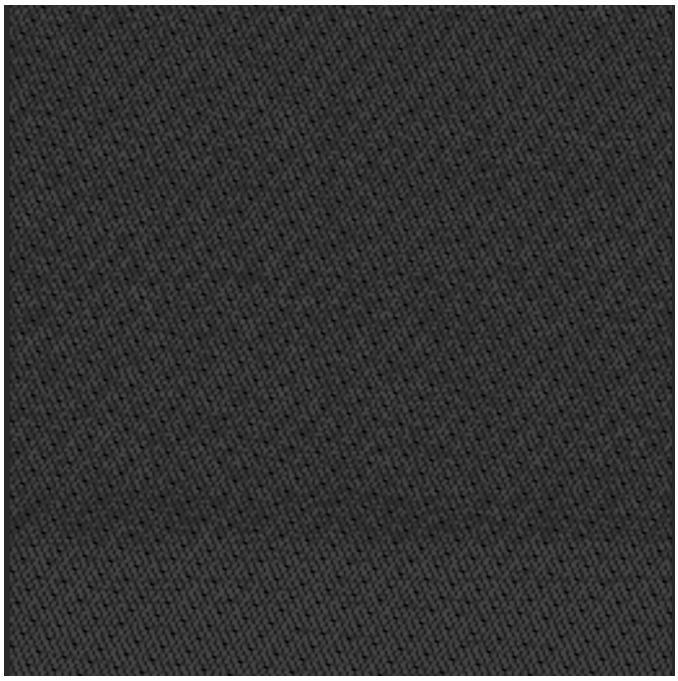

Fig. 5: Second example of the images of malware taken from label 3.

On running the experiment 4 times, these are the results we achieved.

```
Initializing input data normalization.
|=========================================================================================================|
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Validation  |  Mini-batch  |  Validation  |  Base Learning  |
|         |             |   (hh:mm:ss)   |   Accuracy   |   Accuracy   |     Loss     |     Loss     |      Rate       |
|=========================================================================================================|
|       1 |           1 |       00:00:14 |        5.47% |       24.28% |       2.2184 |       2.1084 |          0.0100 |
|       4 |          50 |       00:01:08 |       76.56% |       82.18% |       0.6379 |       0.6393 |          0.0100 |
|       8 |         100 |       00:02:05 |       86.72% |       88.74% |       0.4198 |       0.4079 |          0.0100 |
|      11 |         150 |       00:03:00 |       92.19% |       90.10% |       0.3148 |       0.3699 |          0.0100 |
|      15 |         200 |       00:03:56 |       92.97% |       90.35% |       0.1729 |       0.3288 |          0.0100 |
|      18 |         250 |       00:04:52 |       97.66% |       90.72% |       0.1003 |       0.3460 |          0.0100 |
|      22 |         300 |       00:05:48 |       99.22% |       91.71% |       0.0431 |       0.3524 |          0.0100 |
|      25 |         350 |       00:06:43 |       97.66% |       92.95% |       0.0625 |       0.3274 |          0.0100 |
|      29 |         400 |       00:07:38 |       99.22% |       92.82% |       0.0371 |       0.3604 |          0.0100 |
|      33 |         450 |       00:08:34 |      100.00% |       92.45% |       0.0123 |       0.3347 |          0.0100 |
|      36 |         500 |       00:09:29 |      100.00% |       93.07% |       0.0064 |       0.3618 |          0.0100 |
|      40 |         550 |       00:10:25 |      100.00% |       92.57% |       0.0039 |       0.3763 |          0.0100 |
|      40 |         560 |       00:10:40 |      100.00% |       92.82% |       0.0041 |       0.3754 |          0.0100 |
|=========================================================================================================|

ans =

    0.9257
```

Fig. 6: The first run got a 92.57% accuracy rate

```
|=========================================================================================================|
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Validation  |  Mini-batch  |  Validation  |  Base Learning  |
|         |             |   (hh:mm:ss)   |   Accuracy   |   Accuracy   |     Loss     |     Loss     |      Rate       |
|=========================================================================================================|
|       1 |           1 |       00:00:06 |       13.28% |       27.97% |       2.1875 |       2.0647 |          0.0100 |
|       4 |          50 |       00:01:00 |       90.72% |       81.81% |       0.5282 |       0.6789 |          0.0100 |
|       8 |         100 |       00:01:55 |       90.62% |       91.83% |       0.3487 |       0.3591 |          0.0100 |
|      11 |         150 |       00:02:50 |       93.75% |       90.97% |       0.1905 |       0.3308 |          0.0100 |
|      15 |         200 |       00:03:46 |       95.31% |       92.33% |       0.1595 |       0.2893 |          0.0100 |
|      18 |         250 |       00:04:41 |       98.44% |       94.43% |       0.0771 |       0.2544 |          0.0100 |
|      22 |         300 |       00:05:36 |      100.00% |       93.19% |       0.0138 |       0.3175 |          0.0100 |
|      25 |         350 |       00:06:31 |       98.44% |       94.18% |       0.0639 |       0.3049 |          0.0100 |
|      29 |         400 |       00:07:26 |       98.44% |       94.06% |       0.0426 |       0.2812 |          0.0100 |
|      33 |         450 |       00:08:22 |      100.00% |       93.69% |       0.0085 |       0.3010 |          0.0100 |
|      36 |         500 |       00:09:17 |       99.22% |       93.94% |       0.0177 |       0.3109 |          0.0100 |
|      40 |         550 |       00:10:13 |      100.00% |       93.94% |       0.0039 |       0.3090 |          0.0100 |
|      40 |         560 |       00:10:28 |      100.00% |       94.18% |       0.0060 |       0.2987 |          0.0100 |
|=========================================================================================================|

ans =

    0.9455
```

Fig. 7: The second run got a 94.55% accuracy rate

```
|=========================================================================================================|
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Validation  |  Mini-batch  |  Validation  |  Base Learning  |
|         |             |   (hh:mm:ss)   |   Accuracy   |   Accuracy   |     Loss     |     Loss     |      Rate       |
|=========================================================================================================|
|       1 |           1 |       00:00:13 |        5.47% |       24.28% |       2.2184 |       2.1084 |          0.0100 |
|       4 |          50 |       00:01:08 |       79.69% |       82.18% |       0.6061 |       0.6457 |          0.0100 |
|       8 |         100 |       00:02:04 |       87.50% |       88.89% |       0.4004 |       0.4504 |          0.0100 |
|      11 |         150 |       00:03:00 |       91.41% |       91.46% |       0.2671 |       0.3088 |          0.0100 |
|      15 |         200 |       00:03:57 |       93.75% |       91.34% |       0.1949 |       0.3083 |          0.0100 |
|      18 |         250 |       00:04:53 |       95.31% |       93.07% |       0.1520 |       0.2644 |          0.0100 |
|      22 |         300 |       00:05:49 |      100.00% |       92.95% |       0.0479 |       0.2582 |          0.0100 |
|      25 |         350 |       00:06:44 |       98.44% |       92.95% |       0.0502 |       0.2833 |          0.0100 |
|      29 |         400 |       00:07:40 |      100.00% |       93.07% |       0.0088 |       0.2627 |          0.0100 |
|      33 |         450 |       00:08:37 |      100.00% |       93.94% |       0.0061 |       0.2625 |          0.0100 |
|      36 |         500 |       00:09:32 |      100.00% |       93.69% |       0.0059 |       0.2805 |          0.0100 |
|      40 |         550 |       00:10:28 |      100.00% |       93.56% |       0.0045 |       0.2891 |          0.0100 |
|      40 |         560 |       00:10:43 |      100.00% |       93.07% |       0.0038 |       0.2921 |          0.0100 |
|=========================================================================================================|

ans =

    0.9332
```

Fig. 8: The third run got a 93.32% accuracy rate

```
|=========================================================================================================|
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Validation  |  Mini-batch  |  Validation  |  Base Learning  |
|         |             |   (hh:mm:ss)   |   Accuracy   |   Accuracy   |     Loss     |     Loss     |      Rate       |
|=========================================================================================================|
|       1 |           1 |       00:00:06 |       13.28% |       27.97% |       2.1875 |       2.0647 |          0.0100 |
|       4 |          50 |       00:01:01 |       88.28% |       83.42% |       0.4692 |       0.5764 |          0.0100 |
|       8 |         100 |       00:01:56 |       89.84% |       89.23% |       0.3956 |       0.3552 |          0.0100 |
|      11 |         150 |       00:02:52 |       93.75% |       92.33% |       0.2243 |       0.3171 |          0.0100 |
|      15 |         200 |       00:03:48 |       94.53% |       91.96% |       0.1486 |       0.3037 |          0.0100 |
|      18 |         250 |       00:04:44 |       98.44% |       92.95% |       0.0835 |       0.3356 |          0.0100 |
|      22 |         300 |       00:05:40 |       99.22% |       91.58% |       0.0231 |       0.3185 |          0.0100 |
|      25 |         350 |       00:06:35 |       98.44% |       93.94% |       0.0543 |       0.3081 |          0.0100 |
|      29 |         400 |       00:07:31 |      100.00% |       93.81% |       0.0075 |       0.3064 |          0.0100 |
|      33 |         450 |       00:08:27 |      100.00% |       93.56% |       0.0088 |       0.3007 |          0.0100 |
|      36 |         500 |       00:09:23 |      100.00% |       93.94% |       0.0063 |       0.3107 |          0.0100 |
|      40 |         550 |       00:10:19 |      100.00% |       93.94% |       0.0045 |       0.3171 |          0.0100 |
|      40 |         560 |       00:10:34 |      100.00% |       93.94% |       0.0043 |       0.3188 |          0.0100 |
|=========================================================================================================|

ans =

    0.9431
```

Fig. 9: The fourth and final run got a 94.31% accuracy rate

Our four test runs concluded in an average of 93.69% accuracy rate.

VI. Conclusion

In conclusion, I believe the 94.31% average accuracy rate is an adequate amount to say this experiment was a success. This proves the concept that a Convolutional Neural Network could potentially be used to successfully identify malware and what kind of malware it is. I believe this technology can be used to one day block malware before it even has a chance to cause harm, even before it is discovered and logged into the large databases of known malware.

References

[1] https://www.cisco.com/c/en/us/products/security/advanced-malware-protection/what-is-malware.html

[2] https://www.mathworks.com/discovery/convolutional-neural-network-matlab.html

[3] Christodorescu, M., & Jha, S. (2004). Testing malware detectors. ACM SIGSOFT Software Engineering Notes, 29(4), 34. doi:10.1145/1013886.1007518