Malware Classification Using Deep Learning

By: Andrew Jones | Data Science Capstone | May 2019

**Malware**– Malware is any software intentionally designed to cause damage to a computer, server, client, or computer network. Malware does the damage after it is implanted or introduced in some way into a target's computer and can take the form of executable code, scripts, active content, and other software.

Table of Contents

[Executive Summary 3](#_Toc8137847)

[Project Dataset 4](#_Toc8137848)

[Exploratory Data Analysis 5](#_Toc8137849)

[Data Preparation 6](#_Toc8137850)

[Network Architecture & Training 7](#_Toc8137851)

[Final Results 9](#_Toc8137852)

[Deployment 11](#_Toc8137853)

[Conclusions 12](#_Toc8137854)

Table of Figures

[Figure 1 - Uneven Classes of Malware Files 5](#_Toc8136045)

[Figure 2 - Hex Data Files 6](#_Toc8136046)

[Figure 3 - Hex to Binary Conversion 6](#_Toc8136047)

[Figure 4 - Compressed Binary Stream as .png 7](#_Toc8136048)

[Figure 5 - Class Training Weights 8](file:///C:\Users\212759542\Documents\Grad%20School\Capstone\Malware%20Classification%20Using%20Deep%20Learning%20v1.1.docx#_Toc8136049)

# Executive Summary

As the world continues to become more digitized, there is a growing need within government agencies & large organizations for Digital Forensics. This sub-field related to Cybersecurity is concerned with investigating how cyber crimes were committed, what vulnerabilities need to be remedied and, in some cases, what prosecution steps can be pursued. In the first piece of the job around diagnosing how cyber crimes were committed, there is a consistent backlog of information to investigate, potentially malicious files to classify and a time intensive process to do so. Most digital forensics today involves an investigation conducted by subject matter experts, utilizing proprietary tools and custom making of “features” that can be used to classify malicious files. This process is expensive and time-consuming meaning there is an opening for improvements.

Hence this project is focused solely on using deep learning techniques to classify malware files. As opposed to traditional tabular features that would be used to classify a malware file, in this case deep learning is utilized because we are classifying the file off its binary representation. Quite literally classifying the machine language 0’s & 1’s in a sequential order (binary stream) for each malware file. This process is beneficial not only because it can be utilized much quicker, but also because of its generalizability to all file types – what file can’t be made into its binary form?

Therefore, this project involves creating 3 different deep learning networks to understand their ability to classify these malware files into 9 different classes. These 9 different classes are derived from a Microsoft data set and are therefore mostly files that affect windows users. The class distribution is not even, so I utilized evaluation metrics such as the F-1 score and modified the training weights to prevent overfitting to the majority classes. This project does not focus on a high degree of visualization or EDA, except where necessary to understand the data files & class imbalance.

In the end I was able to achieve a high F-1 score of ≈.9668 on the most advanced of the 3 networks and was happy with my performance. From a personal standpoint, I will not be submitting this algorithm for any type of recognition or financial benefit (Microsoft does still accept admissions to the contest behind this data). However, my benefit is purely on the learning side.

***Please note*** *– While all the code is my own, this paper was heavily inspired by an academic paper on the same subject coming out University College Dublin and can be found* [*here.*](https://arxiv.org/ftp/arxiv/papers/1807/1807.08265.pdf)

# Project Dataset

As mentioned in the executive summary the purpose of this project is to classify known malware files into 9 different classes using the binary representation of those files. These classes are as follows:

**Ramnit** - is a Computer worm affecting Windows users. It was estimated that it infected 800 000 Windows PCs between September and December 2011.

**Lollipop** - This adware program shows ads as you browse the web. It can also redirect your search engine results, monitor what you do on your PC, download applications, and send information about your PC to a hacker.

**Kelihos\_ver3** -  botnet that was capable of sending an estimated 4 billion spam messages a day.

**Vundo** - **Vundo**, or the **Vundo** Trojan (also known as Virtumonde or Virtumondo and sometimes referred to as MS Juan) is a trojan on Microsoft Windows that is known to cause popups and advertising for rogue antispyware programs, and sporadically other misbehavior including performance degradation and denial of service

**Simda** - SIMDA is a family of backdoors capable of stealing information such as user names, passwords, and certificates. It steals information via its keylogging and HTML injection routines. It also executes backdoor commands, compromising the security of the infected systems.

**Tracur** - Win32/Tracur is a family of [trojans](https://www.microsoft.com/en-us/wdsi/help/antimalware-security-glossary) that can redirect your web searches. They do this to earn revenue for the malware authors via online advertisement fraud.

**Kelihos\_ver1** - botnet that was capable of sending an estimated 4 billion spam messages a day.

**Obfuscator.ACY** - This threat has been "obfuscated", which means it has tried to hide its purpose, so your security software doesn't detect it. The underlying code can have almost any purpose.

**Gatak** - Gatak is known for infecting its victims through websites promising product licensing keys for pirated software.

However, because of how we represent these files all exactly the same as a binary image (see next page for example), we are not required to understand these different classes to complete the project. This ability for generalization and use by non-cybersecurity experts is a key benefit of this project. The dataset obtained is approximately .5TB and is comprised of over 21K known malware files, however not all of them will be truly useful after preprocessing. The dataset can be found [here.](https://www.kaggle.com/c/malware-classification/data)

# Exploratory Data Analysis

While performing extensive EDA was certainly not the focus of this project, there were some basic aspects of the dataset I need to understand in order to effectively construct the Deep Learning Network. These essentially take the form of “things to look out for” so that I don’t commit any errors. These things were:

#### Class Imbalance

As seen below there is a high degree of class imbalance in this dataset. The most frequent malware file type (Kelihos\_ver3) is over 70 times more frequent than the least frequent malware file type (Simda). Therefore, we will need to take extra measures to ensure the Neural Network doesn’t overfit to the majority classes.

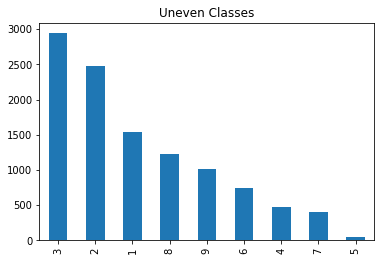


Figure 1 - Uneven Classes of Malware Files

#### Errors in the Malware Files

In order to get the data into binary form we need to overcome errors in the data files themselves. All of the data files received are originally represented as Hex data (this is expanded on in the next section). However, many of the characters within the files were apparently not encoded correctly and just go the ?? designation. This is a not a Hex representation that can be utilized so these bytes had to be simply disregarded. There was a good amount of back & forth on my end testing the preprocessing script that prepared the data due to this problem.

# Data Preparation

To offset the limited focus on Exploratory Data Analysis, there is a high degree of data preparation needed to correctly implement this project. As mentioned in the previous section the data files are originally received in Hex format, not binary. For reference the Hex data looks like this:

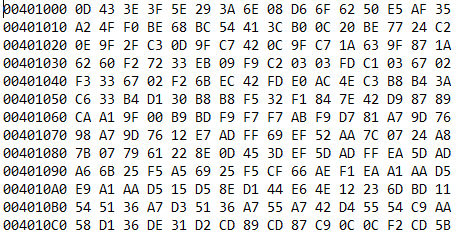


Figure 2 - Hex Data Files

The left most column is the line number, while the other columns have two characters each representing the Hex encoding. This is a base 16 representation that can be used to compress a binary file. For example, each character pair represents one byte or 8 series of bits. A graphic of this conversion is produced below

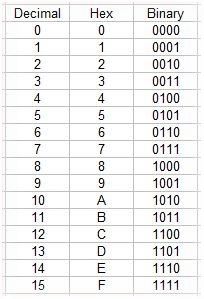


Figure 3 - Hex to Binary Conversion

So, my first task was to translate the hex data into binary data and append that binary stream into a very long array (e.g. list in Python). However, the issue with having this very long string of binary numbers was that they are of unequal length for each file. E.g. Some Binary Streams were millions of digits long while some could be much less. This would not allow for me to utilize my GPU for parallel processing of the files and would make the project time infeasible.

Therefore, the next step in the process is to compress the binary list into a standard 1 X 10000 array. This should preserve the pattern or sequence present within the binary file that we’re trying to hinge on to classify the file. But will allow me to expedite the training process exponentially. To do this I used the Open CV package to resize the array into 1 X 10000. Once the array is resized they look like this:



Figure 4 - Compressed Binary Stream as .png

***Please note – this is just stretched & rotated for display purposes.***

After completing that process for the entire .5TB set of malware files (approximately 3 hours processing time) the data set is ready for training the neural network.

# Network Architecture & Training

The deep learning network architectures used for this project were conveyed in the academic paper to a reasonable level of detail. As mentioned before however, the code was entirely my own. The networks are as follows:

* Convolutional Neural Network (CNN)
* CNN with Long Short-Term Memory (LSTM) Network
* CNN with Bi-Directional LSTM

In the ordering above, each network was essentially an evolution of the previous network. The base part of the network, the CNN, is expected to be appropriate for this classification task as it is used in many image recognition algorithms. Passing the CNN values to an LSTM & Bi-Directional LSTM is expected to increase the networks ability to remember features from further back in the sequence. For example, if within a malware file a function is written at the beginning but is not invoked until later on – this is the type of connection that could matter for classifying the file. The architecture of the models themselves is described in the below table with included reasons behind each aspect.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Network | Layers | Activation | Pooling | Regularization |
| Baseline CNN | **3 convolutional,**  **1 Max Pooling,**  **1 Flatten,**  **2 Dense with 256 cells then 9 (for output)**  This was described in the paper for all types | **RELU for every layer except the last which is SoftMax**  Relu is popular for its ability to resist vanishing gradient. SoftMax is needed for this multiclass scenario. | **Max Pooling with pool size of 64**  This is an area that was not described in the paper. I experimented heavily with this hyperparameter before getting good results. | **L2 regularization on the Convolutional Layers of 0.01. Dropout of .5 after the first dense layer**  L2 regularization reduces overfitting by providing a squared magnitude penalty term on the learning rate. Dropout randomly ignores certain neurons during training. I did not have to determine these rates myself as it was described in the paper. |
| CNN to LSTM | **3 convolutional,**  **1 Max Pooling,**  **1 LSTM with 128 cells,**  **2 Dense with 256 cells then 9 (for output)** | Same as baseline CNN | Same as baseline CNN | Same as baseline CNN |
| CNN to Bi-Directional LSTM | **3 convolutional,**  **1 Max Pooling,**  **1 Bidirectional LSTM with 128 cells,**  **2 Dense with 256 cells then 9 (for output)** | Same as baseline CNN | Same as baseline CNN | Same as baseline CNN |

#### Training Procedure Nuances

In training the network these were some of the specifics not mentioned in the table above:

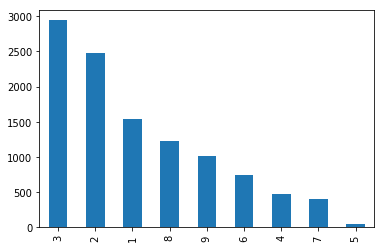
* **Early Stopping with Restore Best Weights Callback** – helps prevent overfitting by retrieving the epoch with the best validation data set F1 score. Will also stop training if the validation F1 score doesn’t improve at least once in every 25 passes *(50 passes for baseline CNN).*
* **Equal Weighting of Classes** – instead of creating a complicated sampling procedure as the authors did, I used the class\_weights parameter of the model.fit call so that instances of the minor classes receive a huge boost to their influence on weight changes. This seemingly does make it “harder” for the network to converge but should make it more generalizable.

Figure 5 - Class Training Weights

Class 3: 0.41018959

Class 2: 0.48699668

Class 1: 0.78668695

Class 8: 0.98271806

Class 9: 1.19129099

Class 6: 1.60689451

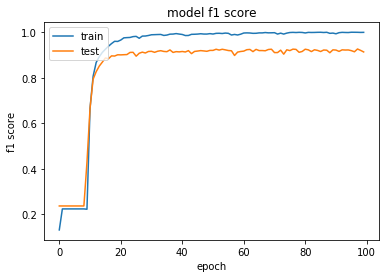
Class 4: 2.5405848

Class 7: 3.03210497

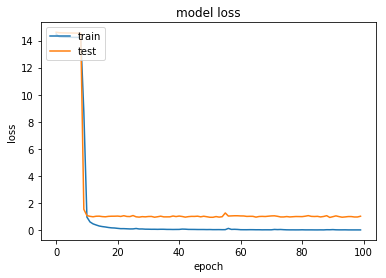
Class 5: 28.73280423

# Final Results

Below are the final results of the different models. As was expected from the onset the CNN with a Bidirectional LSTM had the best performance of all the models. I was very pleased with the results achieved in the models. They were right in line with the scores achieved by the authors of the paper.



#### Baseline CNN



Epoch 98/100 7276/7276

* 15s 2ms/step -

loss: 0.0170 –

acc: 0.9989 –

f1\_m: 0.9991 –

precision\_m: 0.9993 –

recall\_m: 0.9989 –

val\_loss: 0.9710 –

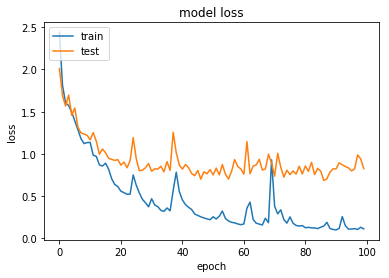
val\_acc: 0.9238 –

**val\_f1\_m: 0.9258 –**

val\_precision\_m: 0.9322 –

val\_recall\_m: 0.9197

#### CNN with LSTM



Epoch 100/100 7276/7276 –

36s 5ms/step –

loss: 0.1105 –

acc: 0.9746 –

f1\_m: 0.9747 –

precision\_m: 0.9787 –

recall\_m: 0.9709 –

val\_loss: 0.8239 –

val\_acc: 0.9565 –

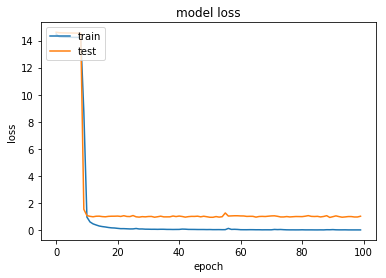
**val\_f1\_m: 0.9577 –**

val\_precision\_m: 0.9607 –

val\_recall\_m: 0.9548

#### 

#### CNN with Bi-Directional LSTM



Epoch 98/100 7276/7276

49s 7ms/step –

loss: 0.0763 –

acc: 0.9864 –

f1\_m: 0.9868 –

precision\_m: 0.9888 –

recall\_m: 0.9847 –

val\_loss: 0.7646 –

val\_acc: 0.9668 –

**val\_f1\_m: 0.9685 –**

val\_precision\_m: 0.9714 –

val\_recall\_m: 0.9657

# Deployment

While not actually implemented as part of this project, I do understand that the steps required to deploy these models should not be neglected. In the real world this can often be where projects fall apart if the infrastructure or organizational readiness just isn’t there. Below are some of the procedures I would recommend for a rather simplistic implementation of the model.

* **Train the model on AWS Sagemaker (or similar)** – this cloud hosted Jupyter Notebook instance makes it easy to train the model in the cloud without crippling your workstation. You would need to ensure the instance was connected to a compute instance that had a GPU available however.
* **Save the trained model back into AWS S3 –** using the AWS SDK for Python you can save the trained model back in an S3 instance right form the Jupyter notebook.
* **Open up an API Endpoint for the model –** once hosted on S3, an API endpoint can be configured to allow users & programs to ping the model for a response. It would be a simple request, response structure. In its simplest form this would allow pre-processed images to be uploaded and a response returned. However, you could also include the preprocessing steps in the program so that it accepts the Hex files in their original form. This would be a longer response time however, so it may be best to separate the two tasks.



# Conclusions

In conclusion some additional value-added aspects of this project that enhance its applicability should be noted:

* **Applications to Cybersecurity defense** – As mentioned several times all files in this particular dataset are known malware files and the goal is to classify them correctly. However, because of the generalizability of the solution, given a dataset that contained malware files & benign files you could use this same type of network to classify along those classes. This would have use as a “first line of defense” for IT systems & networks.
* **Extremely Generalizable** – because the algorithm is working off the binary representation of files, every type of file can be represented in this way. This saves cybersecurity analysts huge effort in manually creating features for analysis.
* **Limited Cyber Domain Experience Needed** – I did not have to work with complicated manual tools or any proprietary software to create features for analysis (e.g. Wireshark)
* **Low Latency** – if implemented as a network defense solution, the mode would need to be optimized for low latency in order to minimize disruption to users / downstream systems needing benign files. The authors of the paper supporting this, achieved a response time of 0.02 seconds for classification of a binary file.

For my final personal views on the project, I was very pleased to simply be able to implement a Deep Learning network on such an interesting domain & dataset. Throughout the project I wanted to limit my own “scope creep” and therefore explicitly did not focus on tasks like Exploratory Data Analysis that may comprise a greater aspect of some projects. I inherently knew that because this would be a challenging project, my focus should be solely on “getting it to work”, even with the help of the academic paper to describe most of the steps. It was a great learning experience for me to be able to configure all of this as it is only my second time utilizing a deep learning architecture for a serious project, and my first time utilizing a network that requires a GPU. In fact, knowing that I wanted to do a Deep Learning project for the Capstone class over Christmas break I built my own Desktop computer that can handle these tasks locally and avoid a layer of complexity on cloud services. Overall it was great experience to gain and I’m excited to have this project in my back pocket for interviews and future resume’s.