CS5242 Project Malware Detection

Group 36
A0105695Y(E0267417) Zhao Mengdan
A0105740R(E0267420) Zhou Shaowen

Introduction

The objective of the project is to train a neural network model to classify whether an exe file is benign by the PE header of it. The main challenge of this task is that the size of PE headers are not fixed, and the number of timesteps of a PE header can reach thousands. To tackle the challenge we first analyzed these headers according to the definition of PE header and extracted the important information from them. Then, we trained three CNN models using the processed headers. Finally, we aggregate the output of these models by taking the average. We achieved the score of 0.99238 and the 8th place by this approach.

Related work

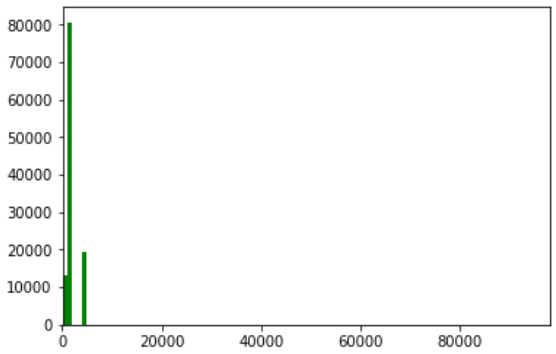
We mainly referred to [1] for the understanding of PE headers, which helped the feature extraction. We referred to [2] for the guidance of designing neural networks for malware detection.

Data Preprocessing

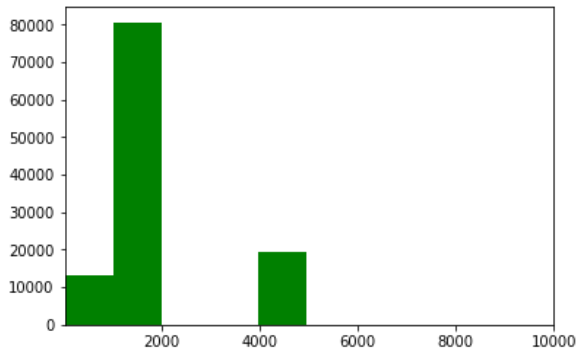
Data Analysis

As the given dataset consists of varying length data entries, we started with data length analysis. The very first approach we tried was padding all data entries to the longest length in the file. However, we quickly found out that training data and testing data have different max_len, where max_len for training data is 98304 while for testing data is 314368. It is not only computational expensive but also results in errors in the model due to the different length. Therefore, we realized that in order for the model to work on both training and testing data, we need to find an appropriate cut-off length such that we can maintain the maximum information and improve computational efficiency at the same time. Therefore, we started with data length analysis.

As shown below, by plotting a length distribution graph for all data entries in training dataset we can see that majority of the data entries are with a length smaller than 10000. And after zooming into range 0 to 10000, we can further observe that the major length distribution is under length 2000.



Length distribution of training data



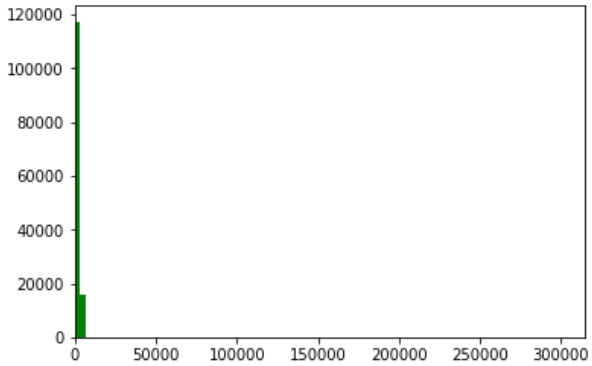
Zoomed-in length distribution of training data

And by further analysis, we observed that there are only 7 data entries out of 113636 in training dataset are longer than 8192. Details can be found in the figure below.

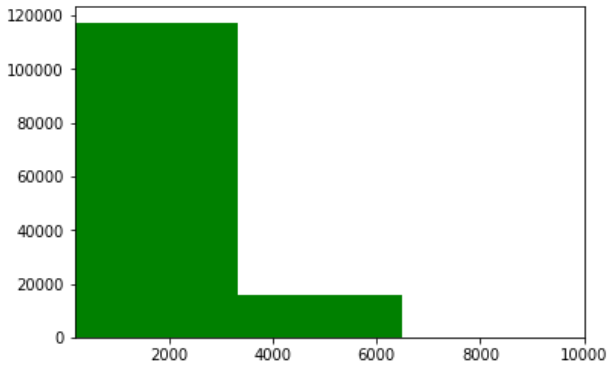
```
In [8]: for l in range(len(X)):
            if (len(X[l])>8192):
                print('index: ', l ,' length: ',len(X[l]), 'corresponding label is: ', y[l])

index:  12628  length:  12288 corresponding label is:  [1]
index:  34871  length:  98304 corresponding label is:  [1]
index:  37170  length:  65536 corresponding label is:  [0]
index:  39998  length:  98304 corresponding label is:  [1]
index:  41975  length:  12288 corresponding label is:  [0]
index:  75599  length:  12288 corresponding label is:  [1]
index:  95385  length:  12288 corresponding label is:  [1]
```

Details of long data entries

To verify our findings, we did a similar analysis for testing data as well and findings are comparable. The longest data in the testing dataset has a length of 314368 while the majority of them are under the length of 8192. Only 5 out of 133223 entries have a length longer than 8192. Therefore, we believed that 8192 can be a good cut-off length to train our data model.



(Length distribution of testing data)



(Zoomed-in length distribution of testing data)

```
In [11]: for l in range(len(testX)):
             if (len(testX[l])>8192):
                 print('index: ', l ,' length: ',len(testX[l]))

index:  25099  length:  65536
index:  72229  length:  12288
index:  92078  length:  314368
index:  117947  length:  297472
index:  124969  length:  44997
```

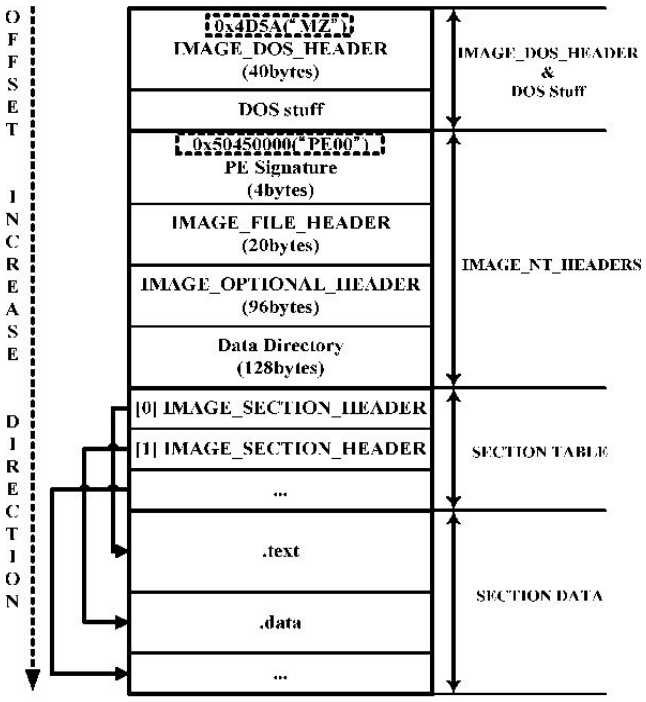
Details of long data entries

Besides this, we also applied a trick during padding. To minimize the impact of padded "0" values to the original data set, we added 1 to all existing byte values before padding such that padded values are differentiated from the original byte values.

As a result, we managed to achieve auc score of 0.98 with CNN model 1 as described in Model Architecture.

Feature Engineering

Feature engineering is the second step we applied to further improve our model prediction accuracy. After doing more online research on PE header structures, we realized that the cut-off data is not in a uniformed shape.



PE Header Structure

As shown in the picture, each PE header file contains 4 sections: DOS Header, Image NT Header, Section Table, and Section Data. The offset of each function is determined in the previous function, which may vary a lot. Therefore, simply cutting all data into the same length might result in information variance. Instead, it is better to locate each of the four sections' offset and trim unused data chunks in between to construct a uniformed dataset.

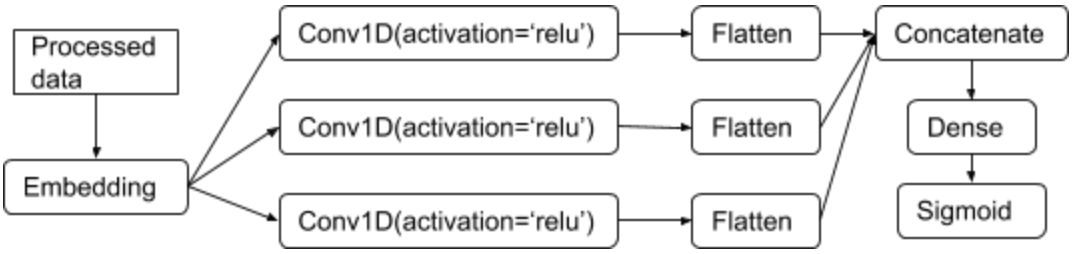
After experiments, we decided to keep all 78 bytes of DOS Header and DOS stub, all Image NT header data, and first 12 bits of all section headers. Since the length still varies for NT headers and Section headers, we also did a length-analysis for them and obtained two magic numbers for padding: MAX_NT = 352, MAX_SEC = 3552.

Visual illustration of the feature engineering process to obtain uniformed dataset can be found in **Appendix** Figure 1 and Figure 2.

Model Architecture

We implemented three CNN models. All the CNN models accept preprocessed data as input, and we use binary cross-entropy as the loss function.

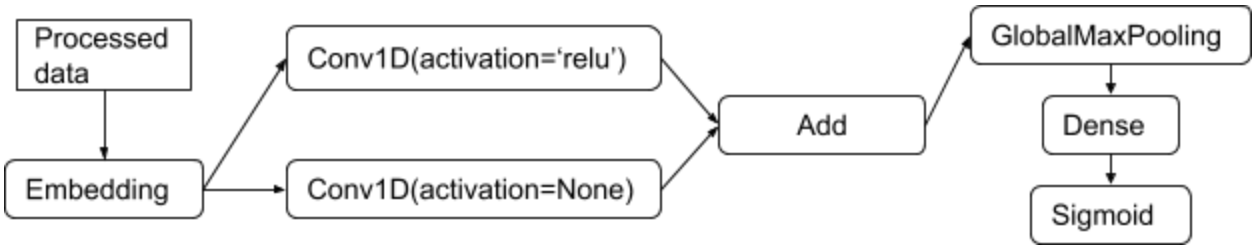
CNN model 1 (CNN ensemble)



Architecture diagram of CNN ensemble

The embedding layer maps each byte to a fixed length feature vector. Then the vectors are passed to three 1D convolution layer with different kernel and stride settings. Malware features may be extracted from these convolution layers. Then each result of 1D convolution is flattened to 1D vectors and concatenated together. Finally, dense layer is applied to the concatenated vectors to obtain the result. ReLU is used as the activation function of dense layers except the last one.

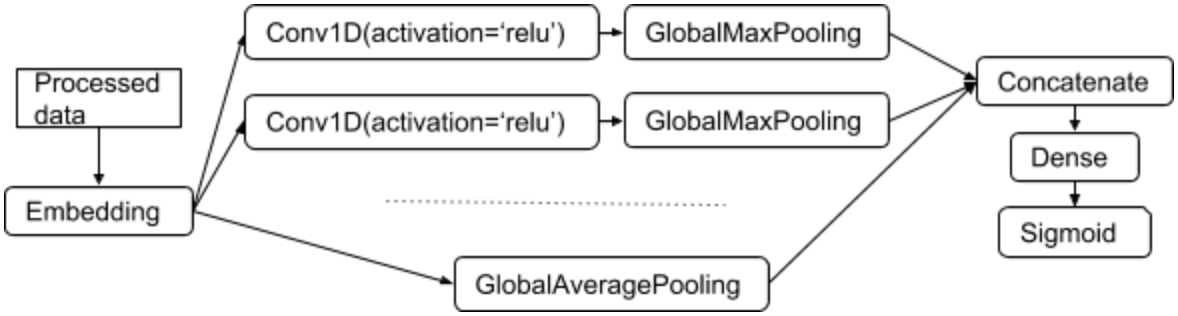
CNN model 2 (MalConv from paper [2])



Architecture diagram of MalConv

The major difference between this architecture and the previous one is that the convolution block contains only two parallel convolution layers which differ on activation function. The convolution layer without activation function works a bit similar to the "identity shortcut connection" introduced by ResNet.

CNN model 3 (Inspired by CNN Ensemble and ResNet)



Architecture diagram of CNN Ensemble + Identity shortcut

This architecture combines the idea from CNN ensembles and ResNet. Multiple convolution layers are running in parallel, and all convolution layers are followed by a max pooling layer. There is also one "identity shortcut connection" besides the convolution layers. Experiments show that this architecture performs a bit better than previous models on processed data in terms of auc score. However, it is still not comparable to the auc score achieved by taking the average of the predictions made by three models.

On Failed Architectures

We first tried basic MLP and CNN models. MLP model can achieve validation accuracy of 0.7 and CNN model can achieve validation accuracy of 0.8.

Inspired by Kolter and Maloof[3], we also experimented with N-gram structure. Based on the hypothesis that malware files are always highly sequence-related, N-gram can be applied in DNN to simulate the relationships of the continuous sequence of n-sized byte items and it's malware classification. As a result, though N-gram model indeed improved the testing accuracy significantly, the computational cost required to train the entire dataset is huge. We were only able to run the N-gram model on 10% of the training dataset before encountering OOM error. Therefore, we decided to give up on this model due to physical constraints.

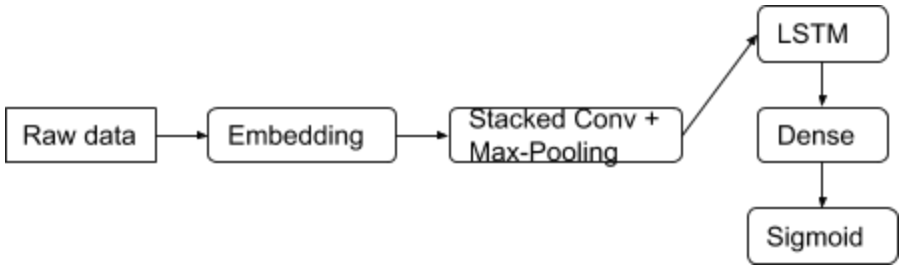
We also tried ResNet. We replaced 2D convolution with 1D convolution and 2D average pooling with 1D average pooling because data are 1D sequences. Though training accuracy is over 0.9, the validation accuracy is very bad and it is around 0.5. Overfitting is a serious issue. Therefore, we think that ResNet may be too complex for the training dataset, and we should keep using shallow CNN models.

RNN is commonly used for classifying sequences of various length, and we tried using LSTM directly on word embeddings. This approach is failed because the length of most sequences is larger than one thousand. The experiment shows that LSTM cannot learn anything useful from long sequences. Both training and validation accuracy remain very low around 0.6 for the first ten iterations, and validation accuracy does not improve over time.

Architecture diagram of naïve LSTM

To handle the long sequences, we then tried truncating sequences. We used only first 256 bytes of PE headers. Accuracy is not improved. The reason may be that too much data which contains crucial information is lost. We tried another technique to cut each long sequence to a group of fixed length short sequences and assign the same label to them. The validation accuracy is still not high enough because a large volume of false positive and false negative data are generated by this approach.

Later, we tried to use convolution layer as feature extractor before LSTM. This time it works. Both training accuracy and validation accuracy reach around 0.97. However, it still does not perform as well as CNN model mentioned before. It achieved auc score around 0.9951 on raw data, which is about 0.0005 worse than those achieved by CNN + feature extraction that we used for the contest. Feature extraction does not improve this model much. However, this model is robust because it performs significantly better on raw data than CNN models alone. The reason may be that the LSTM layer helps resolve the issue of content misalignment.



Architecture diagram of CNN-LSTM

Experiment

Experiment Settings

We configured both local and Google Cloud environment for experiments. Intel i7-7700HQ CPU, 16GB memory, and Nvidia GTX1060 6GB GPU are used for the local environment. For cloud environment, 4 vCPU, 16GB memory and one Nvidia K80 GPU are used. We split the training dataset and used 80% of it for training and the rest for testing. Validation dataset is the same as the test dataset.

Experiment Results

To reduce the chance of overfitting, we added dropout layer after convolution layer and dense layer, and early stopping that monitors validation loss. We also used “adam” as the optimizer. For all models, max pooling performs much better than average pooling.

For Model 1

We tested various padding and stride for convolution layers and find that padding 400 with stride 400, padding 500 with stride 500 and padding 600 with stride 600 performs better overall. We also found that dropout rate of 0.5 is optimum for each convolution layer. This model achieved an accuracy of 0.97254367 and auc score of 0.99570495 on validation dataset.

For Model 2

We found that no dropout layer after each convolution layer performs better. This model achieved an accuracy of 0.97285167 and auc score of 0.99535836 on validation dataset.

For Model 3

Our goal of this model is trying to catch the feature different from previous models, so we set convolution padding and stride to be 2 2, 4 4, 8 8, 16 16, and 32 32. We also found that dropout rate of 0.2 is optimum for this network. This model achieved an accuracy of 0.97131166 and auc score of 0.99577724 on validation dataset.

For Aggregated Model

Model aggregation achieved auc score of 0.99699900 on validation dataset.

Conclusion and Future Works

The max pooling outperforms average pooling for the malware detection task might show that max pooling better captures location-invariant features. Feature engineering which aligns different syntax parts of PE headers can boost the performance of CNN models. CNN plus LSTM model is more robust that it performs better on raw data than CNN models alone. Aggregating multiple models can push the performance limit of a single model.

In the future, we want to develop a model that can outperform current solution without feature engineering.

References

[1] "Peering Inside the PE: A Tour of the Win32 Portable Executable File Format," [Online]. Available: https://msdn.microsoft.com/en-us/library/ms809762.aspx. [Accessed 13 April 2018].

[2] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro and C. Nicholas, "Malware Detection by Eating a Whole EXE," 2017. arXiv:1710.09435 [stat.ML]

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

Appendix

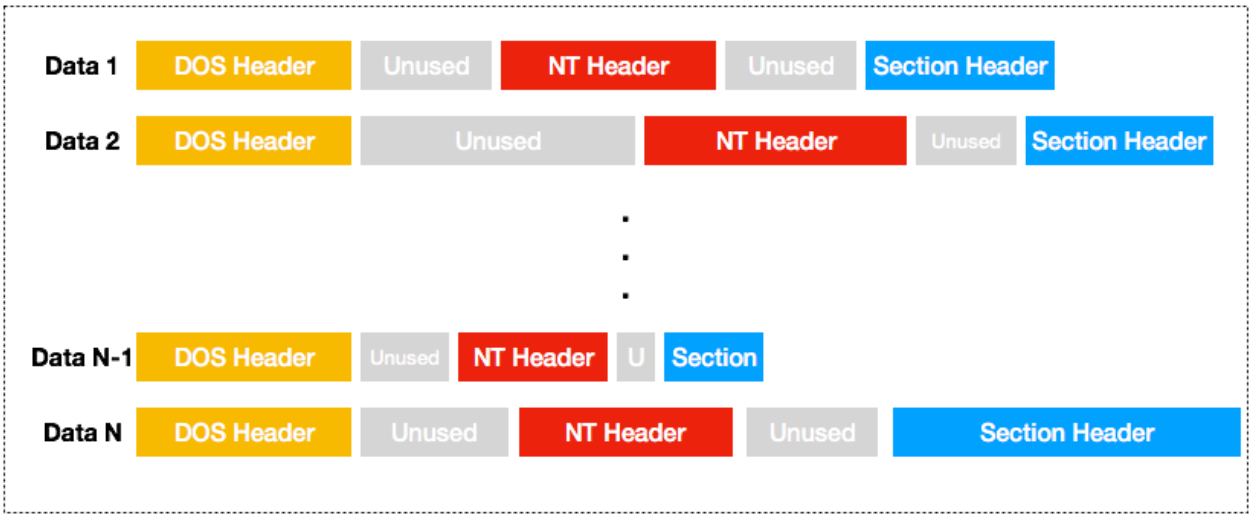


Figure 1. Training data set before feature engineering

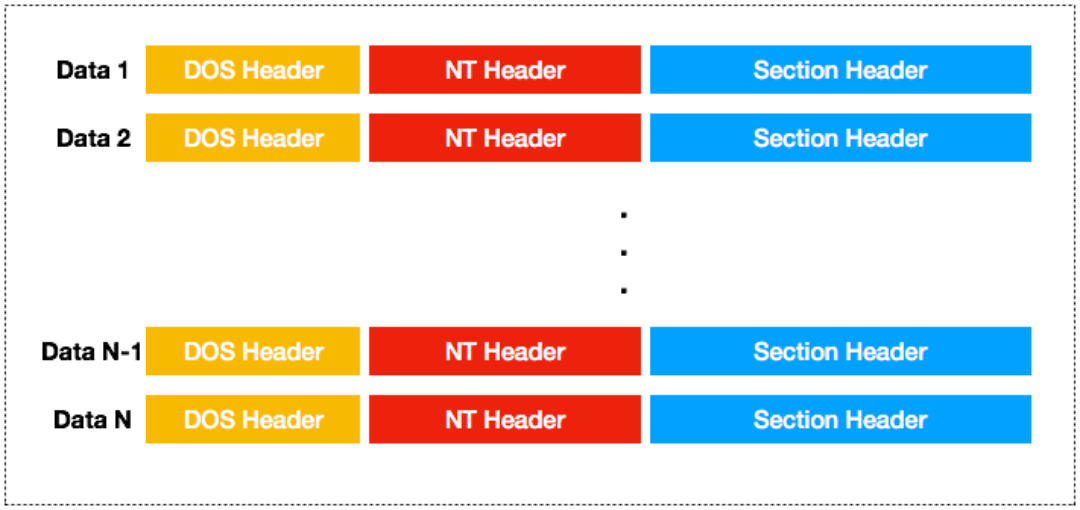


Figure 2. Uniformed Training dataset after feature engineering