Group 48

1. Task description and some highlights of your solution.

- Traditionally, anti-virus and malware detection use signature-based approaches, using pre-determined rules to identify different groups of known malware types. These rules are generally specific and brittle, and usually unable to recognize new malware even if it uses similar functionality.

- Therefore, there have been multiple attempts in using machine learning and deep learning for malware detection purposes. In this task, we only have access to the PE Header of the program. Our solution comprises of approaches where domain knowledge is required, and other approaches that does not require domain knowledge.

2. Related work and model introduction

We have built various models, some of which uses machine learning and others deep learning methods. This paper[3](pages 33-38) gives a very good background about the current popular approaches for malware detection using machine learning and deep learning. However, most of the current approaches require domain knowledge. The baseline approach that does not require domain knowledge is byte n-gram analysis. Nonetheless, in our project, we decide to do use deep learning instead for no-domain-knowledge approaches.

3. Experiment including the setting (machine and dataset partitioning), result analysis (performance of different hyper-parameters, models), discussion of possible improvements.

A.Domain-knowledge Approach: The data given are raw PE byte values which have variable lengths, which meant that although many of the different files have the same data, they are not in the same columns. Thus we embarked on a task of parsing the raw bytes to form a structured representation of the data.

The PE header can be split into four different sections, namely the ms-dos stub, signature, coff file header and section table.

We find that the ms-dos stub is very different from one file to another, and thus there is no lossless way to have a fixed length representation. Since each byte is an assembly code, this meant that all ms-dos stub, regardless of length, could be summarised in a 256 length array, each representing the occurrence of that instruction in the stub.

The other variable length component is the section table. After a brief inspection, we find that most files have at most 26 headers. This was however, not the case for the test data, with quite a few outliers having 30, 40 or more sections. Through our no domain-knowledge approaches we know that the section table contributed the least information gain, thus there was negligible loss in truncating to only 26 sections.

This parsing left us with 4400 columns, more than the original due to extra padding for the section table. If without the section table, we would only need 635 columns. We used this parsed representation with LightGBM, a gradient boosted tree machine learning implementation and obtained

a score of 0.99255 on the public leaderboard, better than deep learning solutions we have tried. The model was trained with a 12 core cpu with 30gb ram in 2 minutes.

We could have improved the score by using another model for the files that the parser failed to parse. Instead, due to a lack of time, we simply gave all failed parse files 0.875, as only ⅛ files were malware.

B. No Domain-knowledge Approach: We investigated 3 different Neural Network Architectures for this class. Neural Networks have performed extremely well for tasks such as Image Recognition and Speech, where the low level features are encapsulated(e.g by the Convolution Layer) and then be used to construct higher level representations. Here, we make use of an EMBEDDING LAYER to encapsulate the automatically encapsulate the lower level features, removing the need for domain-knowledge. It is useful in the analysis of PE Header, not only because the structure is quite complicated, but also because a malicious author can violate the rules and formats of PE Header-which makes the domain-knowledge approach fragile.

B1/ Embedding Layer:

We give the bytes of PE Header directly to the Embedding Layer. Each byte is converted into a feature vector(similar to how word2vec works) of size R^B, where B is the dimension of the feature vector. We experiment with different values for B, and found that B=32 gives better result than B=16,8 or 4. When B=64 or 128; the accuracy is still roughly the same, but the training time becomes much slower.
By using the Embedding Layer, the Lower Level Features are automatically extracted from the PE Header without the need for domain knowledge.

B2/ Multi-layer Perceptron

We will use the output of Embedding Layer.
Here is the architecture of MLP:

```
Dense(256, activation='elu', activity_regularizer=regularizers.l1_l2(0.0001)
Dropout(rate=0.5)
BatchNormalization()

Dense(128, activation='elu', activity_regularizer=regularizers.l2(0.0001))
Dropout(rate=0.5)
BatchNormalization())

Dense(64, activation='elu', activity_regularizer=regularizers.l2(0.0001))
Dropout(rate=0.5)
BatchNormalization()

Dense(1, activation='sigmoid', activity_regularizer=regularizers.l2(0.0001))
```

As can be seen above, we use a lot of regularization(dropout of rate 0.5, regularizer inside the Dense Layer, and even Batch Normalization layer does have effect on regularization as well). As expected,

the more regularization helps the generalization accuracy improve, but the training time becomes slower. Besides that, Batch Normalization also helps improve the training time.

For each Dense Layer, we make use of activation Exponential Linear Unit(ELU), which is a successful extension to ReLU. ELU not only helps prevent vanishing gradient problem, but also in contrast to ReLUs, ELUs have negative values which allows them to push mean unit activations closer to zero. Zero means speed up learning because they bring the gradient closer to the unit natural gradient.

Also, we add L1 Regularizer to the first layer. The L1 norm has been found to be effective when dealing with high dimensional problems and induces sparsity, and the dimension of our input is very large - (time_step, embed_size), which is (4096, 32). We chose 4096 because all except a tiny amount of outliers exceed the length.

The prediction of this model on Kaggle is around 98.7.

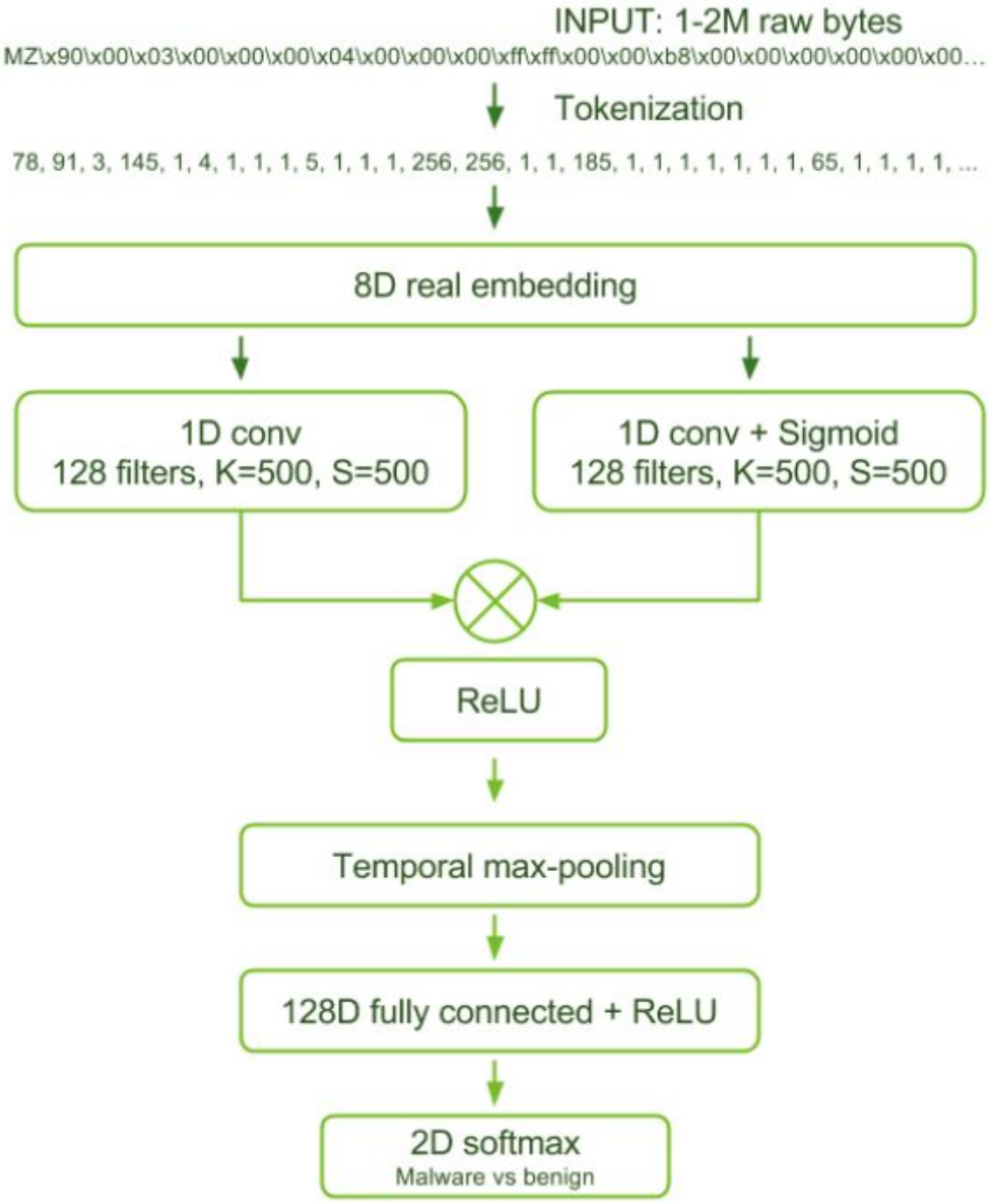
We have also tried another version where, instead of using the first 4096 bytes, we use the 971 columns that our LightGBM model used as feature branches. This sped up the training time significantly, and gave a similar score to the 4096 bytes version. However, the mlp method did not beat the LightGBM model it depended on.

B3/ Convolution Neural Network

- There are also many attempts in using CNN for analysis of the PE bytes. But no attempt has been made on using CNN for PE Header Byte so we want to give it a try.

- First, we arrange the 4096 bytes of each PE Header into a 2D array and visualize the image(Which you can find more in visualization.ipynb). However, the images look very similar for malware and benigh files. We try running different architectures of CNN but it gets very bad result.

- Then, we find this paper[1] and attempts to implement it in Keras(in Convo_custom.py).



Since the 2D image looks too similar, we try with 1D CNN instead of 2D CNN. However , the input of this paper is raw bytes from the entire PE file instead of the header. So, we decide to find a smaller parameter for each layers. The 1D convs have 128 filters, kernel size of 5 and stride of 2. Here are a few design considerations:

- For each convolution layer, we have parallel convolution layer with sigmoid activation. Then we take the elementwise product of the outputs from these two layers The result is passed to the reLU

non-linearity. This enables the sigmoid convolution layer to filter the information allowed out of the reLU convolution layer, which adds additional capacity to the model's feature representations in an efficient way.

- We also attempt to use a Global Max-Pooling Layer. The gated convolutional layer will identify many local indicators of the PE Header, then this layer will assess the relative strength of all of them and create a global combination.

Unfortunately, this works extremely bad in our case. The accuracy barely increases over iterations, no matter how we fine-tune the hyper-parameters. One of the possible reason, in my opinion, is because for PE Header it lacks the local property(for example: CNN works for images they have spatial locality, CNN works for time-series data because they have temporal locality). Two adjacent bytes in the PE Header can have very different meaning and not related to each other at all.

B4/ LSTM With Attention Layer[2]

We use Long-Short Term Memory Network to see if it can encapsulate the high level features from the low level features and sequential property. If RNN(or LSTM) works well, then it would be extremely useful because RNN can handle the nature of variable length sequences, so less domain knowledge is required to truncate all inputs to same length. Also, thankfully we are using just the PE Header, so the timestep length is not too long and LSTM can handle this easily without the fear of vanishing gradient.

Our model has 3 LSTM Layer stacked together, with a Dropout of 50% at the end. We use Adam as the optimizer scheme.

Then, the output will be given to an Attention Layer. The motivation of using Attention Mechanism is this: Usually, only the output of the last hidden unit of LSTM will be used for the prediction. Although that output are related to values from earlier hidden units, those relations are not very strong. So, bytes at the end of PE Header will always have much more influence to determine malicious files than the bytes at the beginning of PE Header. On the other hand, Attention Mechanism, basically, will take all the outputs of each hidden unit of LSTM, then will decide which parts to pay more ATTENTION to.

We determine the final prediction after the attention layer as following:

$$x = (h_0 + h_1 + h_2 + \ldots + h_{n-1}) / n$$
$$a_i'' = v^T \tanh(A*h_i + B*x)$$
$$a_i = \exp(a_i) / (\exp(a_0) + \exp(a_1) + \ldots + \exp(a_{n-1}))$$
$$\text{final\_prediction} = a_0*h_0 + a_1*h_1 + \ldots + a_{n-1} * h_{n-1}$$

Here, a_i is the weight for each hidden unit, which indicates how much attention should be paid for that hidden unit. Please note that x is the average of all hidden units, so we want each weight to not only takes in its hidden unit but it should have an overview of all hidden units as well.

For this implementation, the main drawback is the extremely slow training time. We did not foresee this happening, so we can only train for 4 days and get a result around 0.9. Compared to MLP or LightGBM approach, this approach is too slow in both training and testing phases.

C. Ensemble

We try to ensemble the the MLP and LightGBM with raw bytes, using weighted vote ensemble.

So, if MLP predicts the value y_1 and LightGBM predicts the value y_2, then our ensemble will return $y = a * y_1 + b * y_2$

So, we want to find a and b such that the loss function of y will be minimized.
In order to do that, we run the algorithm through a simple MLP of 1 layer(in file ensemble.py). By training an MLP of accuracy 0.987 and a Lightgbm with accuracy of 0.99034, we managed to get an ensemble of accuracy 0.9915 on Kaggle.

4/ Workload Assignment

Here is how we split the work:
Together: Data exploration, visualisation.
Li Kai: In main charge of the domain-knowledge and no domain-knowledge approach using lightGbm. He also built the feature parser for PE Header.
Le Trung Hieu: In charge of no-domain-knowledge approaches.

It is just a general overview of our workload. We discussed all the ideas. Li Kai also helped me fine-tune the models.

Reference:

[1]: https://devblogs.nvidia.com/malware-detection-neural-networks/
[2]: https://arxiv.org/pdf/1709.01471.pdf
[3]:http://www.covert.io/research-papers/deep-learning-security/Convolutional%20Neural%20Networks%20for%20Malware%20Classification.pdf