Android Malware Detection using Sequential Convolutional Neural Networks

XingPing Sun1*, JiaYuan Peng1, HongWei Kang1, Yong Shen1

1School of Software, Yunnan University, Kunming, Yunnan, 650091, China
*Corresponding author’s e-mail: sunxp@ynu.edu.cn

Abstract. As the largest Operating System in the smart phone market, Android is gaining more adherents in recent years, which makes it increasingly important to detect Android malware correctly and efficiently. In this paper, we proposed a way to detect it with sequential convolutional neural networks. In the pre-processing work, Android package files were dissected into Dalvik operation codes. These codes were treated as text sequence in our designed model. This model was trained on 3000 samples, and tested on two large datasets. One with 10,174 samples where our model reached an accuracy of 90% in 1.2 seconds and the other with 20,083 samples where its accuracy was 88% and cost 2.4 seconds.

1. Introduction
According to the report published by 360 secure cyber centre and its laboratory in 2007 [1], the number of people infected with Android malware has been reduced to about 500,000 everyday in China thanks to those proposed methods using traditional machine learning algorithm to detect Android malware [2, 3]. It’s still a large number though.

Deep learning started its fashion since Hinton proposed auto encoder in 2006 [4]. It does perform surprisingly well on certain tasks that we cannot define a function to calculate its result. Among those deep learning networks, convolutional neural networks (CNNs) have been applied to many fields ever since it reached a state-of-art accuracy on ImageNet Large Scale Visual Recognition Competition (ILSVRC) [5].

A github project dealing with Chinese text classification gave us inspiration [6]. It treated every Chinese character as an id, not a one-hot vector, and thus sequential convolutional neural networks was used to extract the features of text. This less-space-cost and less-time-cost method managed to reach an accuracy of more than 90% when testing on the dataset offered by THUCNews [7].

Using some currently popular reverse-engineering tools, apktool and dedexer for example, an apk (Android package) file can be easily translated into Dalvik (Android virtual machine) operation codes [8]. These codes, which range from 0 to 226, can be treated as an id itself. In this way, every apk file was dissected into an id sequence, and our model used sequential convolutional neural networks to learn its features and then classified it.

2. Pre-processing

2.1 Dissecting apks
Every apk file is a zip-formatted archive file where there is a dex file. A dex file contains all the java source code of an Android application. By unzipping the apk file, we can easily get the dex file. And then apktool (an android reverse-engineering tool) is used to get the smali files of this dex file. Each
A smali file represents a java class. To ‘translate’ them into Dalvik operation codes listed in [8], another reverse-engineering - dedexer - is used here. By concatenating these operation codes of every smali files together, we can get the operation code sequence of an apk file.

2.2 Cutting and padding

Next, to be trainable for the network, for those sequences generated in the first step, we need to make their length equal. In other words, we have to define a max length that every sequence can have. Assuming the length we defined is $k$, for sequences whose length is longer than $k$, we cut them from the beginning; that is to say, we only took the last $k$ codes. And for shorter sequences, we padded them with ‘0’ at the end of them to make them as long as $k$ (‘0’ means no operation in Dalvik operation code). An experiment was carried to ascertain the value of max length which will be shown in the hyper-parameter section. Actually we also tried to cut the longer sequences from the end and pad the shorter ones with ‘0’ at the end. It turns out that they can perform well but they need a larger value of max length, because there are some similar operation code patterns at the beginning of these sequences which contain less information than other parts of them.

3. Network structure

There are 5 layers in our network - an embedding layer, a convolutional layer, a max pooling layer and 2 fully-connected layers. The weights of every layer are learned jointly in the training process. What we need to adjust is the value of hyper-parameters in each layer which will be described in detail in the following paragraphs. For each hyper-parameter, we carried an experiment to ascertain its value which will be shown in the hyper-parameter section.

3.1 The embedding layer

The embedding layer is first layer of our network whose input is a fixed-length-sequence of an apk file after pre-processing. As mentioned before, the operation codes ranging from 0 to 226 can be treated as ids, thus the sequence can be seen as a vector. It can be sparse since we cannot guarantee the length of every apk file’s operation codes is no shorter than the max length we defined before. Thus an embedding layer is designed here to make the vector dense before it is inputted into the next layer. In the hyper-parameter section, we also found that the model with an embedding layer performs much better than one without such a layer.

The embedding dimensionality is a hyper-parameter we have to define in this layer. Assume the length of the input vector is $k$, and the dimensionality is $d$. Multiplying the input vector by a matrix of size $k \times d$, a vector of length $d$ we get here is the output of this layer. This matrix multiplication does not change the relationships among the vector; the points used to be close in the original vector can remain to be close. That’s to say the embedding layer projects the input vector into a dense one and reserve its inner relationships as well.

3.2 The convolutional layer

A convolutional layer is the second layer of our network whose input is the projected dense vector outputted by the embedding layer. This layer is designed to extract the features of the vector and detect some code patterns a malware may obtain.

The number of filters, the size of kernel and the step length of kernels are the hyper-parameters of this layer. The experiments to ascertain their values will be shown in hyper-parameter section.
Figure 1. Convolutional layer group example 1.

Figure 2. Convolutional layer group example 2.

Figure 3. Edge example.

Figure 4. Padding example.

Figure 5. A convoluted vector.

The input vector will be divided into several groups whose size is the size of kernel. And the number of groups is determined by the kernel size, the step length of kernels and the length of the vector itself. For example, in figure 1 the length of vector is 7, the kernel size is 2 and its step length is 1, there will be 6 groups. The same vector in figure 2, the kernel size changes to 3 and its step length changes to 2, there will be 3 groups this time. However, in some cases, the end of a vector may not be included. Like it is shown in figure 3, the same situation as what is in figure 2, but the vector’s length is 8. We can see that the last number of the vector is not in any group. Two solutions can be used to solve this problem. One way is to pad it with a certain number at its end like in figure 4. Actually this solution is used to make the edge numbers as included as other part of the vector. Take figure 1 (where the edges are included) as an example, numbers from index 2 to 6 are included in at least 2 groups while the edges (index 1 and 7) are only included in one group and the information they carry may not be well used. In this case, padding the vector with a certain value at its end and beginning will help the convolutional layer consider every part of the vector equally. The other way is to simply ignore the edges. When the edges contain little useful information, it can also be an efficient solution. We tried both the two solutions and their results will be shown in the hyper-parameter section.

Each of the divided group will multiply by several vectors whose length is the size of kernel and the number of vectors is the number of filter (the weights of these vectors are jointly learned in the network). Figure 5 shows the convoluted result of the vector shown in figure 2 when the number of filters is 3 and we name these three vectors $v_1$, $v_2$, $v_3$. And this convoluted vector is the output of this layer.

3.3 The max pooling layer

This layer is designed to subsample the features extracted by the convolutional layer. Like what we do when we read code, we can often simply skip a method’s conditions and read its core code to figure out the method’s function.

The size of pool is a hyper-parameter of this layer and also the experiment to ascertain it will be shown in the hyper-parameter section.

The convoluted vector outputted by the previous layer is the input of this layer. It will also be divided into several groups whose size is the size of pool. And the max value of each group will be selected to form the output of this layer. Figure 6 and figure 7 are two examples with the pool size of 2 and 4.
3.4 The max pooling layer

Finally, we put two fully connected layers at the end of our networks. The first one is supposed to learn the relationships among the features outputted by the previous max pooling layer. The second one is to classify the vector which output the probability of input data to be malware and non-malware.

\[ u = \text{ReLU}(v_i W_1 + b_1) \]

\[ y = \text{softmax}(v_o W_2 + b_2) \]

where \( v_i \in \mathbb{R}^m \), is the input of this layer; \( W_1 \in \mathbb{R}^{m \times u} \), is the weight matrix of this layer; \( b_1 \in \mathbb{R}^u \), is the bias of this layer; \( v_o \in \mathbb{R}^u \), is the output of this layer.

The input of the second fully-connected layer is the \( u \)-length vector outputted by the former fully connected layer, as it is shown in figure 9. Multiplying it by a matrix of size \( u \times 2 \), we here get a 2-dimension vector. This 2-dimension vector will be projected by another activation function (softmax is used in our case) to form the output of our entire networks. Its two dimensionality indicates the probability of the input vector to be malware and non-malware respectively. And it can be given by:

where \( v_o \in \mathbb{R}^u \), is the output of this layer (also the output of last layer); \( W_2 \in \mathbb{R}^{u \times 2} \), is the weight matrix of this layer; \( b_2 \in \mathbb{R}^2 \), is the bias of this layer; \( y \in \mathbb{R}^2 \), is the output of this layer.

4. Hyper-parameters

As mentioned in the previous section, there are 7 hyper-parameters we need to define. To ascertain a value for each of them, we tried a series of values varying in a range. And to measure the goodness of each value, we designed a simple network (which is actually part of our model). The goodness of a value mainly depends on the accuracy and the loss the network can get when a hyper-parameter is set to this value in the testing dataset. In the embedding case, we also take the time efficiency into consideration which will be shown in its part. The network may change a little to ascertain each hyper-parameter, and the every network structure will be shown in their respective part, too. The network was trained on a dataset with 617 malware and 668 non-malware. It was tested on a dataset with 1512 malware and 1612 non-malware. All the testing data are not in the training data. We trained
and tested each value for 10 times to get its average accuracy and average loss which are used to measure its goodness.

4.1 Max length of each sequence
As mentioned in the pre-processing section, we need to define max length each sequence can have. To ascertain this value, we changed it from 100 to 8000. The step of the change can be as large as 1000 in the earlier trial and it gets smaller as the range gets smaller. And then we designed a simple network with 2 fully connected layers to classify the vectors we input. The input vector is a operation code sequence to be cut into a certain length in the way we mentioned in the pre-processing section. And we choose the value of a relatively higher accuracy and a relatively lower loss. The step length is 1000 at first, as it is shown in figure 10. And it got lower losses at 1000, 3000 and 7000, but it got the highest accuracy at 1000. Next, we decreased the step length to 100 and tried some value smaller than 1000. It got the lowest loss and the highest accuracy at 400. Again, we decreased the step length to 10, and tried some value between 300 and 500. And it turned out 400 is the best choice for max length.

![Figure 10. Experiment result to ascertain a value for max length.](image1)

![Figure 11. Experiment result to ascertain a value for embedding dimensionality.](image2)

4.2 Embedding dimensionality
Here, we added an embedding layer to the network we designed before to ascertain the max length. Since the max length is set to 400 finally and due to the function of embedding layer is to project the input vector and make it dense, we tried values ranged from 2 to 200. In figure 11, besides the accuracy and the loss of each value, it also shows the time cost to test every 200 samples whose mathematical unit is millisecond. We can see from the picture that the accuracy does not change a lot when the dimensionality is set to values larger than 100, but it takes a lot more time. Thus we set it to values smaller than 100 with smaller steps. And 16 is a better value with high accuracy, low loss, and good time efficiency. Also, comparing figure 10 and figure 11, we can see that it reaches a higher accuracy just by simply adding an embedding layer to the network.

4.3 The convolutional layer's parameters
The convolutional layer has three hyper-parameters, they are the number of filters, the kernel size, and the length of kernel step. Besides, we also need to compare the results of padding and non-padding as we mentioned in the convolutional section. We added a convolutional layer between the firstly fully connected layer and the embedding layer. To ascertain a hyper-parameters’ value, other ones should set to a fixed value. Their results are respectively shown in table1, table2, and table 3. The value we chose is written in red. With these three hyper-parameters set to the fixed values of best result, that’s 16, 5, and 1 as we showed in the table, we tried it two more times on a padding and a non-padding one. The result is shown in table 4.
4.4 Pool size
By adding a max pooling layer between the first fully-connected layer and the convolutional layer, a new network structure is built to ascertain a pool size. Its result is shown table 5.

4.5 The number of hidden units
We do not need to change the structure of network this time. We set values ranged from 8 to 1024. The result shows in figure 12. It has the highest accuracy and the lowest loss at value 256.

| Table 1. Experiment result to ascertain a value for the number of filters. |
|-------------------|---|---|---|---|---|---|---|---|
|                 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 |
| **acc**         | 0.77 | 0.78 | 0.79 | 0.79 | 0.79 | 0.79 | 0.79 | 0.79 |
| **loss**        | 0.45 | 0.42 | 0.40 | 0.39 | 0.40 | 0.43 | 0.49 | 0.47 |

| Table 2. Experiment result to ascertain a value for the kernel size. |
|-------------------|---|---|---|---|---|---|---|---|
|                 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| **acc**         | 0.79 | 0.80 | 0.79 | 0.79 | 0.79 | 0.79 | 0.79 | 0.79 |
| **loss**        | 0.40 | 0.40 | 0.40 | 0.38 | 0.39 | 0.40 | 0.40 | 0.40 |

| Table 3. Experiment result to ascertain a value for the kernel step. |
|-------------------|---|---|---|---|---|---|---|---|
|                 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| **acc**         | 0.79 | 0.78 | 0.79 | 0.77 | 0.77 | 0.77 | 0.78 | 0.76 |
| **loss**        | 0.39 | 0.41 | 0.42 | 0.44 | 0.47 | 0.47 | 0.47 | 0.48 |

| Table 4. Experiment result for padding. |
|-------------------|---|---|
|                  | pad | non-pad |
| **acc**         | 0.78 | 0.79 |
| **loss**        | 0.57 | 0.56 |

| Table 5. Experiment result to ascertain pool size. |
|-------------------|---|---|---|---|
|                 | 100 | 200 | 300 | 400 |
| **acc**         | 0.80 | 0.82 | 0.77 | 0.80 |
| **loss**        | 0.44 | 0.39 | 0.45 | 0.40 |

Figure 12. Experiment result to ascertain a value for hidden units.

5. Experiment

5.1 Environment
Our experiment was carried on a Macbook Pro whose CPU is Intel core i5 and memory is 8GB.

5.2 Dataset
Our malware dataset is offered by VirusShare.com, and non-malware dataset is downloaded from one of the largest Android application store – 360 app store.

5.3 Result
We trained our network on a dataset with 1516 malware and 1612 non-malware. And it was tested on two datasets. It performs well on both of the dataset within a short time. Detail results are shown in table 6.

| samples | malware | non-malware | accuracy | loss | time(s) |
|---------|---------|-------------|----------|------|---------|
| 10174   | 5123    | 5051        | 0.9029   | 0.36 | 1.2     |
| 20083   | 10063   | 10020       | 0.8800   | 0.45 | 2.4     |

Acknowledgments
This work is funded by the Open Foundation of Key Laboratory of Software Engineering of Yunnan Province under Grant No. 2015SE204. Thanks for the dataset offered by VirusShare.com and 360 app store, without which we can hardly carry out these experiments.

References
[1] 360 Secure Cyber Center. (2018) Android Malware Report. http://bbs.360.cn/thread-15355463-1-1.html.
[2] Yerima, S. Y., Sezer, S., Mewilliams, G., & Muttik, I. (2013). A New Android Malware Detection Approach Using Bayesian Classification. IEEE, International Conference on Advanced Information NETWORKING and Applications, Vol.12: 121-128.
[3] Sahs, J., & Khan, L. (2012). A Machine Learning Approach to Android Malware Detection. Intelligence and Security Informatics Conference: 141-147.
[4] G. E. Hinton and R. R. Salakhutdinov. (2006). Reducing the dimensionality of data with neural networks. Science, 313(5786):504-507.
[5] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. International Conference on Neural Information Processing Systems, Vol.60: 1097-1105.
[6] Gaussian. (2017) Text Classification with CNN and RNN. https://github.com/gaussian/text-classification-cnn-rnn.
[7] Maosong Sun, Jingyang Li, Zhipeng Guo, Yu Zhao, Yabin Zheng, Xiance Si, Zhiyuan Liu. (2017). THU Chinese Text Classification. http://thuctc.thunlp.org.
[8] Gabor Paller. (2008) Dalvik Opcodes. http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html.