Deep learning apply in malicious code classification

Heqing Yin
Department of information science and technology
University of Science and Technology of China
yhq123@mail.ustc.edu.cn

Abstract:

Nowadays , as the rapidly development of Internet ,the kind of malicious software , or malware ,is exponentially increasing and spreading .surely , various automated tools and methods are used by malware authors widely .therefore ,we can use tools to extract the common feature which able to distinguish various kinds of malicious code from different malware family

This article propose a new method for malicious code classification by extracting operate code sequences and applying sentence classification to these operate code sequences

The experimental results show the proposed method can effectively distinguish malware families

Keyword: computer security convolutional neural network deep learning text classification

摘要: 今天，随着互联网技术的快速发展，恶意代码呈现指数爆炸的态势增长。无疑，各种自动化代码生成工具被恶意软件作者广泛应用是其中原因之一，因此，我们同样可以运用工具去提取这些代码中能够用来区分不同源头的特征。这篇文章通过实验把深度学习应用到代码文件分类从而提出一种新的恶意代码分类方法。在文章最后的实验结果当中显示了这是一种卓有成效的分类方法。

关键词：计算机安全，卷积神经网络，深度学习，文本分类

1. 导言

早期的反病毒软件大都单一的采用特征匹配的方法，简单的利用特征串完成恶意代码的识别与分类，随着恶意代码技术的发展，恶意代码开始在传播过程中进行变形以躲避查杀[1]，此时同一个恶意代码的变种数量急剧增加，形体较本体也发生了较大的变化，反病毒软件已经很难提取出一段代码作为恶意代码的特征码。在这种情况下，广谱特征码随之诞生[2]，广谱特征码将特征码进行了分段，通过掩码字节对需要进行比较的和不需要进行比较的区段进行划分，然而无论是特征码扫描还是广谱特征，都需要在获得恶意代码样本后，进行特征的提取，随后才能进行检测，这使得恶意代码的查杀具有一定的滞后性，始终走在恶意代码的后面。为了针对变种病毒和未知病毒，启发式扫描应运而生。启发式扫描利用已有的经验和知识对未知的二进制代码进行检测，这种技术抓住了恶意代码具有普通二进制文件所不具有的恶意行为，例如非常规读写文件，终结自身，非常规切入零环等等。启发式扫描的重点和难点在于如何对恶意代码的恶意行为进行提取。特征码扫描、查找广谱特征、启发式扫描，这三种查杀方式均没有实际运行二进制文件，因此均可以作为恶意代码静态检测的方法。随着反恶意代码技术的逐步发展，主动防御技术、云查杀技术已越来越多的被安全厂商使用，但恶意代码静态检测的方法仍是效率最高，被运用最广泛的恶意代码查杀技术。这篇文章在 Related works 里面介绍了代码分类领域已有的研究成果，Proposed method 里面介绍了我们提出的方法。Experimental results 把我们的方法和前人的工作进行了对比。Conclusion 总结全文并展望 future work.

2. 相关工作

恶意代码分类和监测都方法有很多种，一般来说都是根据恶意代码的特征来进行分类，根据特征的来源分

类方法一般分为以下几种

- 基于二进制代码：最简单直接的静态特征就来自于二进制代码文件，在文本分类当中，最通用的特征之一就是 N-gram,并且被应用到恶意代码分类当中 Kephart, Sorkin [3]。而对整个代码的结构信息有一个好的概括的一个特征就是 PE Header ,PE Header 包括了这个代码的大小，调用的 API 地址，编译系统等信息，利用 PE Header 提供的结构化信息 • 进行代码分析被证明是非常有效的方法，而且非常容易提取，这个方法由 Wang, Wu [4]提出。若直接把二进制文件当中的二进制序列当成 ASCII 码来处理时，可以根据翻译成的字符串是否有语义来进行分类 Ye, Chen [5]。
- 基于反汇编代码：直接基于反汇编代码的最通用的特征就是操作码，采用 n-gram 或者 n-perm 形式[6]，
- 基于控制流图：调用关系图直接描绘了程序的各个函数间的调用关系，调用关系图不能提供控制流在过程内部流动的信息，因此这是整个二进制代码文件在执行过程中数据流和控制流的一种高层次的表现方式。
- 基于语义：定义语义的一种方式是根据寄存器状态的改变，比如程序开始与结束时寄存器的状态对，如果有着相当大规模的输入输出时可以采用符号化表达[7]。另一种定义语义的方式是通过程序调用的 API 集合，API 调用集合或者踪迹为待检测的二进制文件提供了一个行为轮廓[8]
- 混合特征：为了提高分类的准确度，已有研究人员把不同层次的特征融合在一起形成混合特征，Masud, Khan [9]结合三个不同层次的特征的摘要：二进制的 n-gram，汇编衍生特征，和系统调用。

Majid VafaeiJahan 等人提出了 Role-opcode 方法 Ghezelbigloo and Vafaeijahan [10]，简要的来说，Role-opcode 方法就是给每个 opcode 分配一个 Role,提取出 opcode 的 n-gram 后用各自的角色来代替 opcode,而后选取不同的特征用 Random Committee ,Random Forest， SVM 等机器学习算法来建立模型

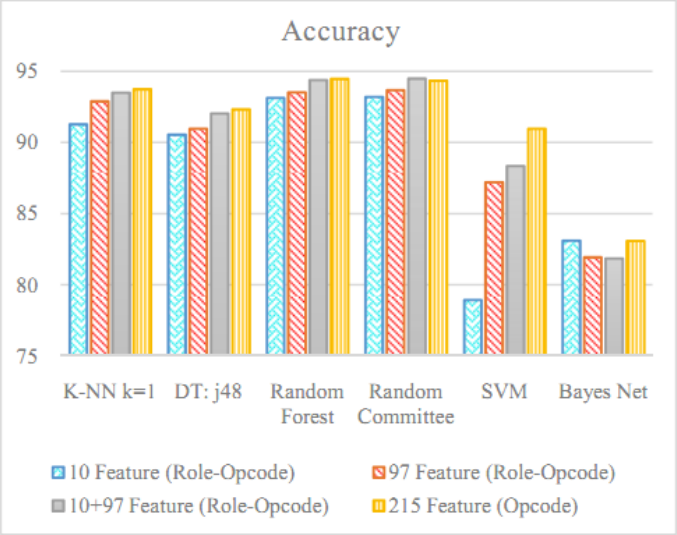


Figure 1 不同分类方法的准确度

基于静态分析的可视化技术也被提出，在[3]中他们先把代码的二进制文件转化成位图，在根据灰度值转化成 entropy graph ，最后在训练过程中进行相似度计算，结果显示同一个家族的恶意代码往往具有相似的 entropy graph ,而且不同的家族也有不同的计算复杂度，这种方法很容易被自动化操作。并且能被用来对大量未知的恶意二进制代码文件进行预处理。

3. Proposed method

1.1 实验目的

用机器学习的方法来进行分类需要提前指定特征，而随着互联网的发展，恶意代码的变种也越来越多，越来越难以发现他们之间的界限，新兴的深度学习在处理这类特征模糊的分类问题上的优势便凸显出来了。当然在利用卷积神经网络进行文本的分类领域也不乏前人的研究 Zhang and Wallace [11]，

因此用深度学习来进行恶意代码的分来也算是我在学习了 CNN 之后的一个实践。

1.2 实验原理

为了介绍我们的方法，首先简要介绍一下什么是 CNN，以及它的作用。

CNN: Convolution Neural Networks 是人工神经网络的一种。它的权值共享网络结构使之更类似于生物神经网络，降低了网络模型的复杂度，减少了权值的数量。该优点在网络的输入是多维图像时表现的更为明显，使图像可以直接作为网络的输入，避免了传统识别算法中复杂的特征提取和数据重建过程。卷积网络是为识别二维形状而特殊设计的一个多层感知器，这种网络结构对平移、比例缩放、倾斜或者共他形式的变形具有高度不变性。

而我们用它来对句子进行分类，因此，这里进行的是一维的卷积运算，而卷积核的选取又能决定提取出来的特征和运算复杂度，在进行训练之前，还要对句子进行预处理，主要目的是把单词替换成向量。而后初始化权重矩阵和 bias，训练过程是不断地抽取输入样本，由公式计算出对应的估计标签，并于实际标签作比较，根据偏差由定义好的优化算法对 wights ,bias 进行调整（一般都是用梯度下降算法，后向传播算法的一种）。

训练流程：

1. 初始化 Weight,bias 向量(Weight 不能初始化为全零)
2. 输入随机选择的一批训练样本 x,和标签 y
3. 计算 y_{eval}=x*Weight+bias
4. 根据 y, y_{eval},根据选择的误差函数（这里选择交叉熵），计算 loss，当训练步数满足一定的条件时用测试集对网络进行评估，结果写入硬盘
5. 由 loss 的大小，根据选择的优化函数来调整 Weight，bias，回到 2

1.3 操作流程

1.3.1 从文本文件中加载句子

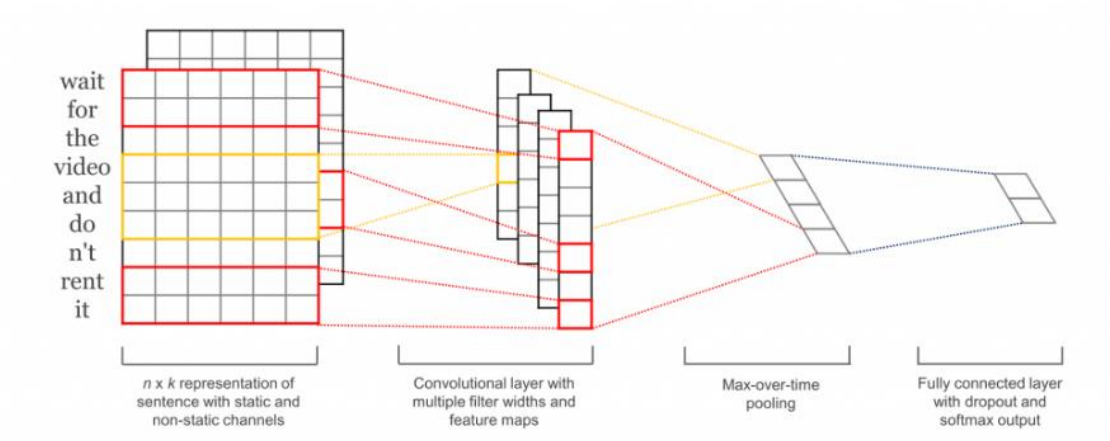
1.3.2 填充/截断每个句子到指定的最大长度

1.3.3 建立词汇索引并且映射每个词到一个整数，这样每一个句子都成为一个整数的向量

1.3.4 输入到 tensorflow,启动训练流程

1.3.5 启动 tensorboard，设置工作路径，查看 tensorboard

模型：要构造的网络模型粗略表示如下



第一层映射词到低维度的向量。下一层在映射的单词向量上用不同的卷积核进行卷积运算，例如，一次移动长度为 3，4，5 的距离。接下来，我们 max-pool 卷积层的的结果到一个长特征向量，加入丢弃规则，

然后用 softmax 层对结果进行分类。

训练结果用交叉熵（cross-entropy）来测量当前网络参数下根据输入的样本推断出来的预测结果与真实标签之间的误差，我们的目标就是用优化方法（optimizer）来最小化 loss

用 10 折交叉验证法来防止过拟合并且提高整个数据集的准确率，训练集和测试集的比例是 9：1

1.4 软硬件环境

1.4.1 Hardware

Processor numbers :24
model name: Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz
cache size: 15360 KB
cpu cores : 6
MemTotal: 132119504 kB
Buffers: 209652 kB
Cached: 7875280 kB
SwapCached: 359592 kB

1.4.2 Software

System CentOS release 6.4(Final)
Python version-2.7.11
Tensorflow version-0.9

Tensorflow :tensorflow 是 google 在 2015 年发布并开源的人工智能系统，利用这个系统可以大大简化我们这个神经网络的编写，省略了很多基础的 coding。TensorFlow 后台计算依赖于高效的 C++，与后台的连接称为一个会话(session)。TensorFlow 中的程序使用，通常都是先创建一个图(graph)，然后在一个会话(session)里运行它。

1.5 数据信息及预处理

我们的实验是基于微软 Kaggle 大数据竞赛提供的数据，微软提供的数据包括训练集、测试集和训练集的标注。其中每个恶意代码样本(去除了 PE 头)包含两个文件，一个是十六进制表示的.bytes 文件，另一个是利用 IDA 反汇编工具生成的.asm 文件。

```
10001000 8B 4C 24 08 56 57 BE 00 60 00 10 8B 11 8B 06 39
10001010 10 75 18 8B 78 04 3B 79 04 75 10 8B 78 08 3B 79
10001020 08 75 08 8B 40 0C 3B 41 0C 74 0D 83 C6 04 81 FE
10001030 18 60 00 10 73 06 EB D5 33 C0 EB 03 6A 01 58 5F
10001040 5E C2 08 00 55 8B EC 81 EC 04 01 00 00 53 33 DB
10001050 39 5D 10 74 07 B8 05 00 07 80 EB 36 8B 45 08 56
10001060 BE 08 02 00 00 57 56 8D 78 14 FF 75 0C 57 E8 3D
10001070 34 00 00 83 C4 0C 8D 85 FC FE FF FF 53 53 68 04
10001080 01 00 00 50 56 57 53 53 FF 15 F0 50 00 10 5F 33
10001090 C0 5E 5B C9 C2 0C 00 55 8B EC 81 EC 50 02 00 00
100010A0 8D 85 B0 FD FF FF 50 8B 45 08 83 C0 08 50 FF 15
100010B0 E8 50 00 10 8D 85 C4 FD FF FF 6A 08 50 FF 75 0C
100010C0 E8 EB 33 00 00 83 C4 0C 33 C0 C9 C2 08 00 56 8D
100010D0 71 18 56 E8 70 37 00 00 59 8B 4C 24 0C 3B C1 76
100010E0 04 33 C0 EB 28 8D 04 09 50 6A 00 FF 74 24 10 E8
100010F0 FC 36 00 00 56 E8 4E 37 00 00 D1 E0 50 56 FF 74
10001100 24 20 E8 A9 33 00 00 83 C4 1C 6A 01 58 5E C2 08
10001110 00 53 8A 5C 24 08 56 8B F1 F6 C3 02 74 35 8B 46
10001120 FC 57 8D 7E FC 8D 0C 40 48 8D 34 8E 78 14 55 8D
10001130 68 01 FF 76 F4 83 EE 0C FF 15 F8 50 00 10 4D 75
10001140 F1 5D F6 C3 01 74 07 57 E8 51 33 00 00 59 8B C7
10001150 5F EB 16 FF 36 FF 15 F8 50 00 10 F6 C3 01 74 07
10001160 56 E8 38 33 00 00 59 8B C6 5E 5B C2 04 00 56 8B
10001170 F1 68 10 04 00 00 6A 00 83 66 14 00 8D 46 18 50
10001180 E8 6B 36 00 00 83 A6 28 04 00 00 00 83 C4 0C C7
10001190 06 E8 51 00 10 C7 46 04 C4 51 00 10 C7 46 08 B4
100011A0 51 00 10 C7 46 0C 98 51 00 10 C7 46 10 80 51 00
100011B0 10 8B C6 5E C3 B8 01 40 00 80 C2 04 00 B8 01 40
100011C0 00 80 C2 0C 00 8B 44 24 04 05 28 04 00 00 50 FF
100011D0 15 E4 50 00 10 C2 04 00 8B 44 24 04 05 28 04 00
100011E0 00 50 FF 15 E0 50 00 10 33 C0 C2 04 00 B8 01 40
100011F0 00 80 C2 1C 00 56 8B F1 E8 14 00 00 00 F6 44 24
10001200 08 01 74 07 56 E8 94 32 00 00 59 8B C6 5E C2 04
10001210 00 C7 01 E8 51 00 10 C7 41 04 C4 51 00 10 C7 41
10001220 08 B4 51 00 10 C7 41 0C 98 51 00 10 C7 41 10 80
10001230 51 00 10 C3 55 8B EC 81 EC 08 02 00 00 53 56 8B
10001240 75 10 57 85 F6 75 0A B8 57 00 07 80 E9 0C 01 00
10001250 00 8B 5D 0C 6A 10 5F 57 68 78 55 00 10 53 E8 0D
10001260 36 00 00 83 C4 0C 85 C0 75 0A 8B 45 08 89 06 E9
"5GpVKiAvyOcJlnaSZkfx.bytes" [dos] 8192L, 475136C
```

Figure 2 Bytes 文件

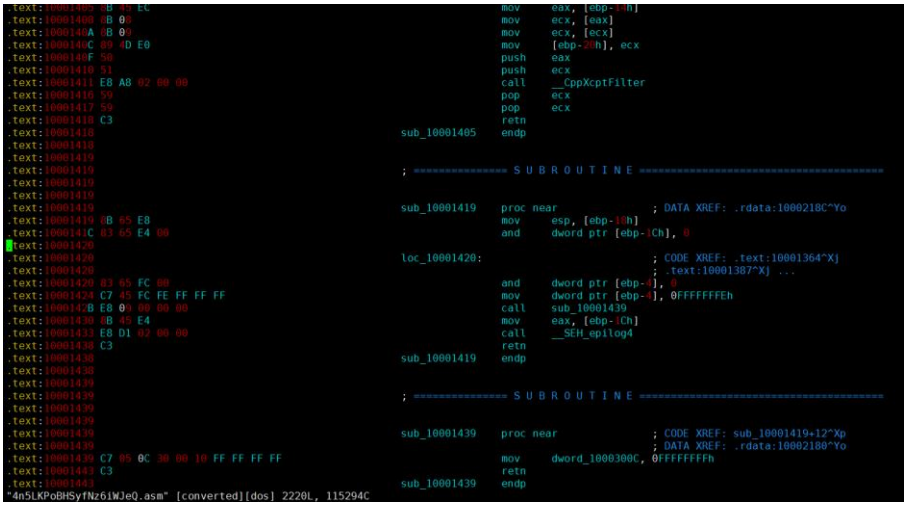
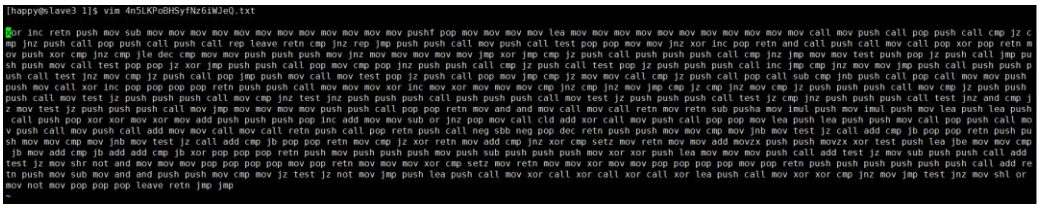


Figure 3 asm 文件

把每一个 bytes 文件当中可执行指令的字节码全部放到一行，每一类当中形成一个文本文件，每一行代表一个 bytes 文件，如下图所示

Figure 4 Bytes 文件提取的句子 txt 文件

把每一个 asm 文件中每一行以.text 或者以 CODE 开头的指令按原来顺寻全部提取到一行，每一类当中形成一个文本文件，每一行代表一个 asm 文件的所有可执行指令，如下图所示：

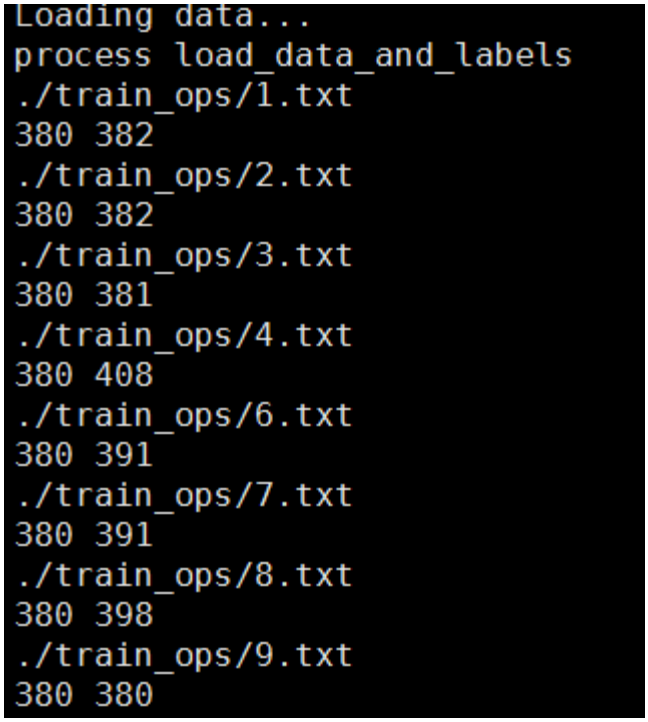


每个标签下的文件个数如下

file	lines
1.txt	1541
2.txt	2478
3.txt	2942
4.txt	475
5.txt	42
6.txt	751
7.txt	398
8.txt	1228
9.txt	1013

Figure 5 每个文件中语句行数（即每一类中文件数量）

注意到每一类的样本数量相差极大，而训练过程中需要每个标签的样本数量上要相等，而参与训练的样本太少则势必影响到验证数据时的准确率，因此选择剔除标签 5，这样就只剩下 8 个标签。又因为有一些代码样本在提取之后没有发现可执行指令，或者没有以'.text'或'CODE'开头的代码段，导致在句子集中为空，根据填充原则，这样的空句子也会被填充到指定长度，而实际上这样的全填充句子在训练过程中不会给神经网络的收敛带来好处。为了提高最后的准确度，这种无效句子会被剔除。由于 8 个标签当中最小的是 7 号只有 398 个样本，选定每一类进入训练集的样本数量是 380，这样

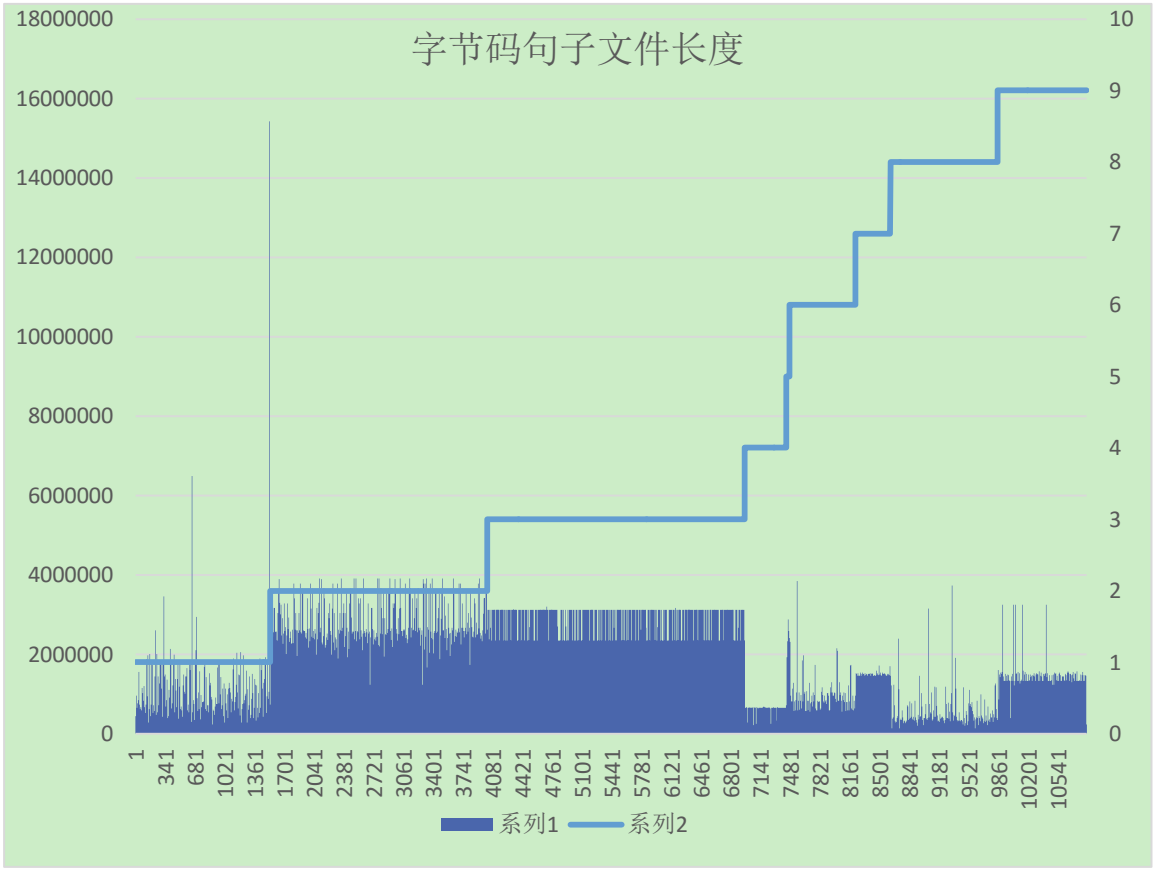


每一类读入样本数与实际输入样本数

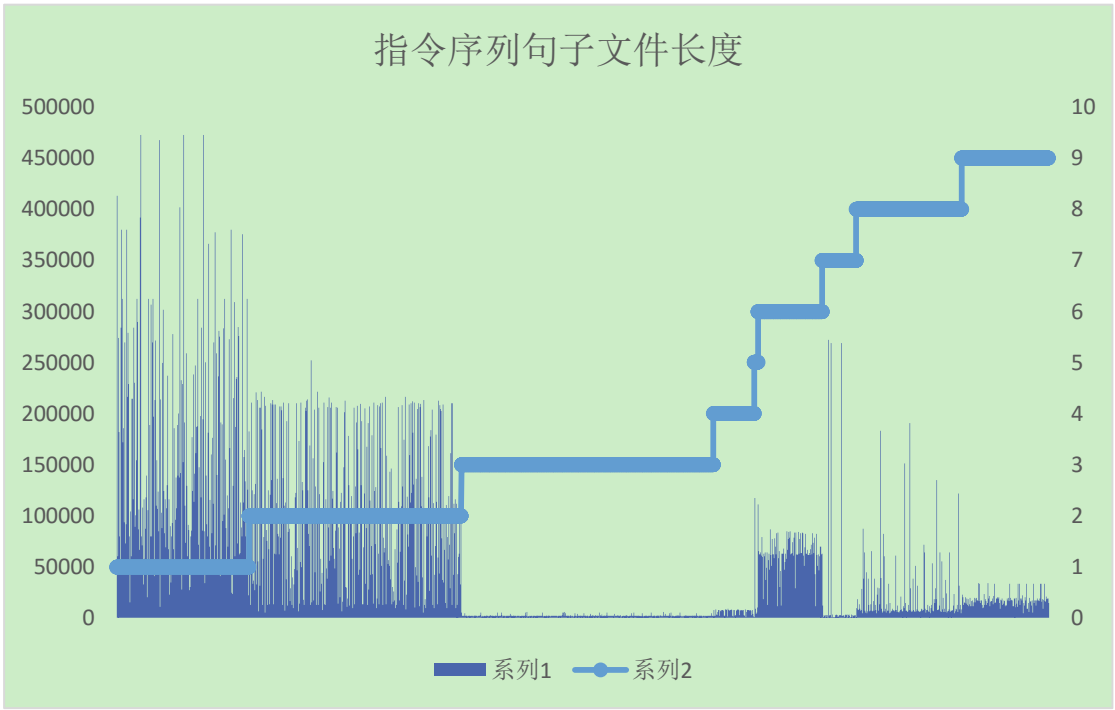
步骤：

对原始数据进行预处理：

对字节码句子文件进行统计发现每一类 bytes 文件当中含有的可执行指令长度相差很大，由于服务器内存太小，只有选择 10000 作为最大长度，超过这个长度的句子进行截断处理，短于这个长度的句子填充符号一直到等于指定长度。



对指令序列句子文件进行统计发现每一类 asm 文件当中含有的可执行指令长度相差很大，用 matlab 计算得出当选定长度为 25000 时能够覆盖 93.5%的文件，但由于服务器内存太小，为提高计算速度，选择 5000 作为最大长度，，超过这个长度的句子进行截断处理，短于这个长度的句子填充符号一直到等于指定长度



4. 实验结果

参数:

Model Hyperparameters		Training parameters:		Misc Parameters:	
embedding_dim	128	batch_size,	32	allow_soft_placement,	True
filter_sizes	16,32,64	num_epochs,	100	log_device_placement,	False
num_filters	128	evaluate_every,	50		
dropout_keep_prob	0.5	checkpoint_every,	50		
l2_reg_lambda	0.5				

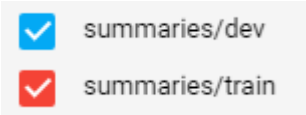
Train 1

Text Type: hex bytes sequence

Sequence_length:10000

MAXLINE:200

Remove empty sentences: false



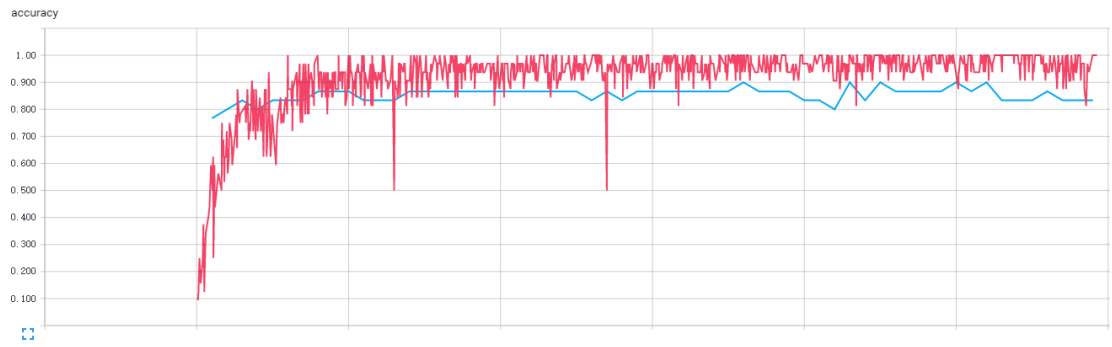


Figure 6 Accuracy-steps

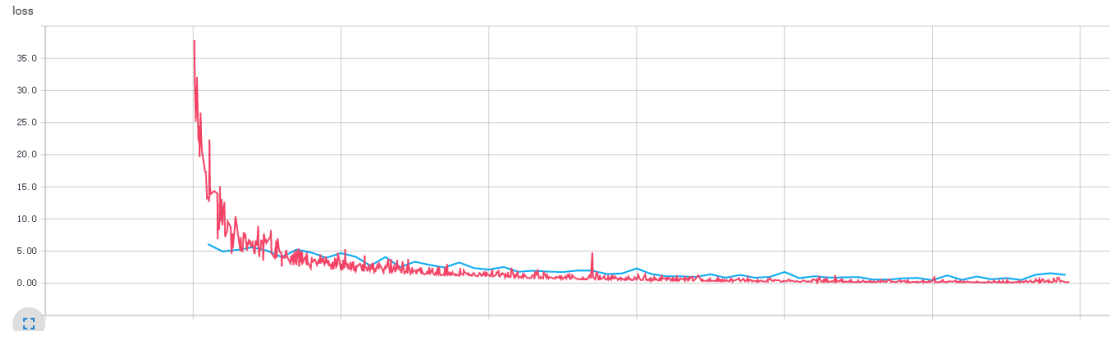


Figure 7 Loss-steps

训练 2850 步用时 38 hours 26 minute，收敛用时 400 steps 左右

平均 0.809 min per step

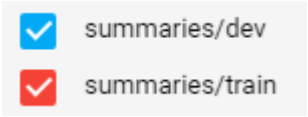
Train 2 对指令序列的训练

Text Type: opcode sequence

Sequence_length:5000

MAXLINE:380

Remove empty sentences: true



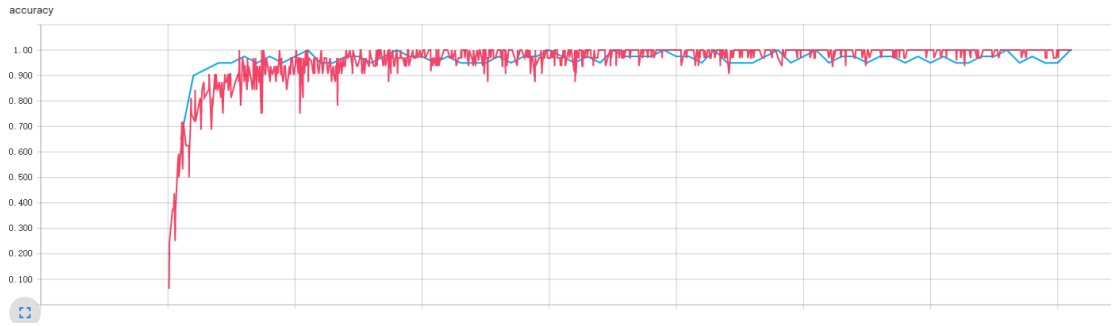


Figure 8 Accuracy-steps

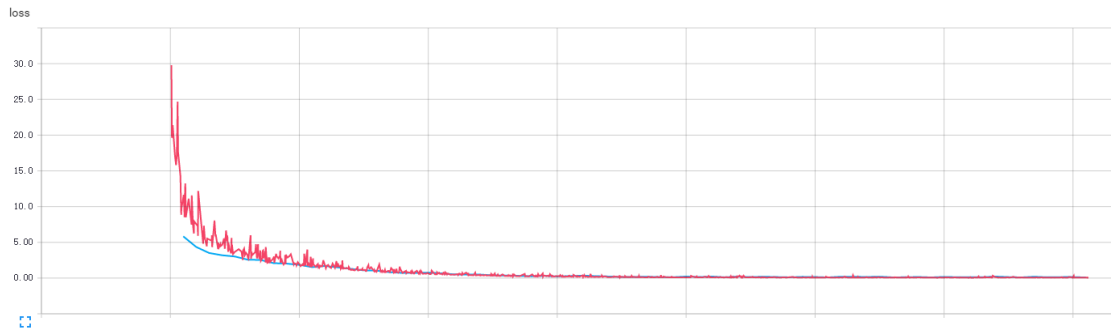


Figure 9 Loss-steps

训练 3952 步用时 28 hours 34 minute，收敛用时 500 steps 左右

平均 0.434 min per step

Train 3

Text Type: opcode sequence

Sequence_length:20000

MAXLINE:390

Remove empty sentences: false

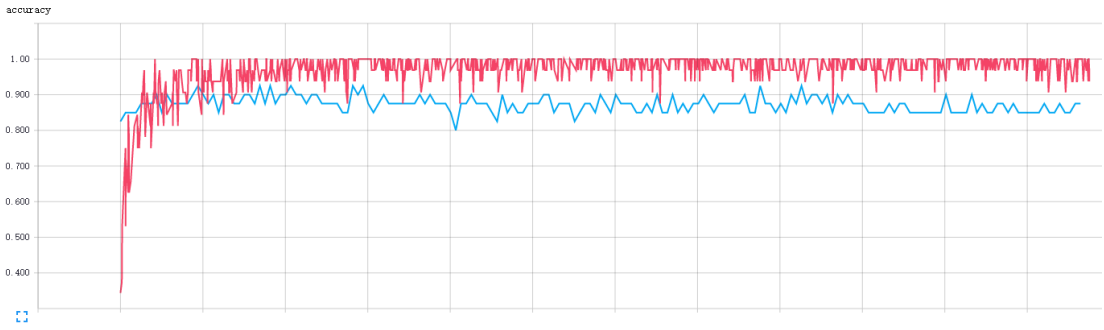


Figure 10Accuracy-steps

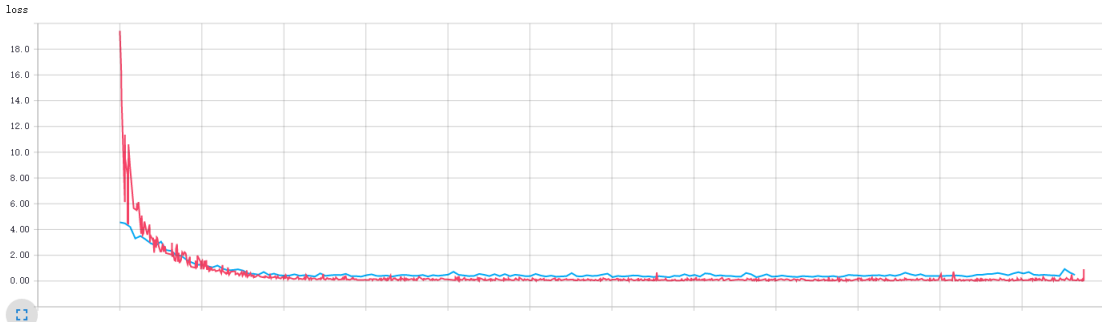


Figure 11 Loss-steps

Accuracy 最后稳定在 0.85 左右，训练用时 9d 16h 左右，完成 9310 steps,平均 1.50 minutes per steps

Train 4

Text Type: hex byte sequence

Sequence_length:5000

MAXLINE:200

Remove empty sentences: false

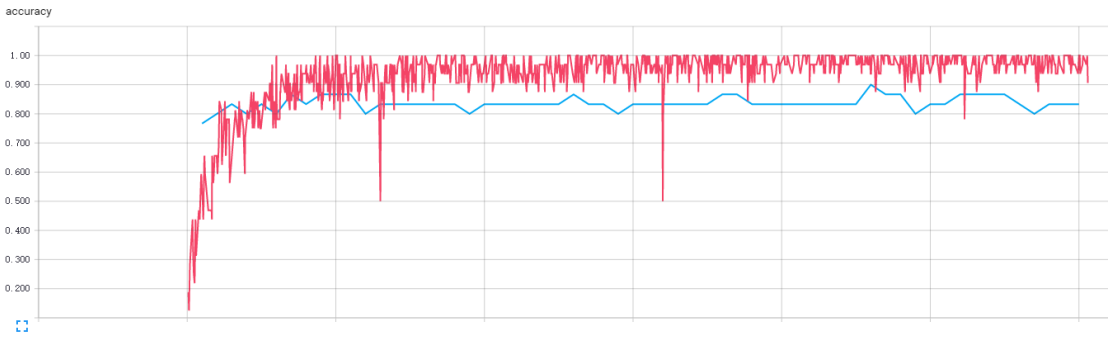


Figure 12Accuracy-steps

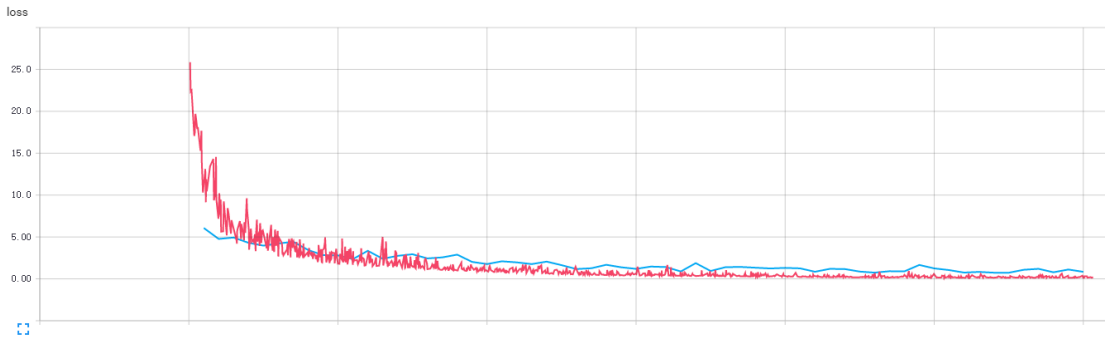


Figure 13 Loss-steps

训练用时 22h 左右，完成 3050 steps,平均 0.43 minutes per steps

5. 结语

由此可知，训练指令序列速度更快，也更快收敛,从图中可以直接看到训练指令序列的准确率能够达到 0.95，而直接训练直接码只有 0.85 的准确率，剔除无意义的空句子要比保留这些空句子训练更快收敛，也更稳定。

相比于先对指令序列提取 n-gram 特征，再用有监督的机器学习的方法，本文提出的方法减少了操作上的复杂度，相比于基于图像的方法，该方法没有消耗大量内存去存储灰度图像素点的需要。深度学习的方法的优势就在于不需要事先提取供机器使用的特征，直接输入样本即可，这对于特征不明显的分类极大减轻了提出特征的工作量。

由于我们实验室的条件有限，没有 GPU 计算资源，因此训练样本容量有限，这样就导致过拟合现象的出现。在有 GPU 集群的条件下，在 Tensorflow 里可以打开 GPU 加速，运用并行计算，可以使得训练速度有很大的提升。由于机器的内存也有限，当设置某些参数过大时甚至会导致系统自动杀死进程，在条件允许的情况下，可以把计算负担分散到多个处理器多线程同时进行。

由于时间仓促，原始数据的预处理阶段完全是采用人工的方式来进行的，费时费力，在将来完全可以依靠程序自动化实现数据的预处理，参数的选择等工作，这样也对计算资源提出了很高的要求

Acknowledge:

数据样本来自 https://www.kaggle.com/

Tensorflow 代码和教程来自 https://www.tensorflow.org/

实验环境提供：中国科学院信息工程研究所 6 室

参考文献

1. Tabish, S.M. and M.Z. Shafiq. *Malware detection using statistical analysis of byte-level file content*. in *Knowledge Discovery and Data Mining*. 2009.
2. Seideman, J.D., B. Khan, and C. Vargas, *Quantifying Malware Evolution through Archaeology.* Journal of Information Security, 2015. **6**.
3. Kephart, J.O., et al. *Biologically inspired defenses against computer viruses*. in *International Joint Conference on Artificial Intelligence*. 1995.
4. Wang, T., C. Wu, and C. Hsieh. *Detecting Unknown Malicious Executables Using Portable Executable Headers*. in *Networked Computing and Advanced Information Management*. 2009.
5. Ye, Y., et al., *SBMDS: an interpretable string based malware detection system using SVM*

ensemble with bagging. Journal in Computer Virology, 2009. **5**(4): p. 283-293.

6. Runwal, N., R.M. Low, and M. Stamp, *Opcode graph similarity and metamorphic detection.* Journal in Computer Virology, 2012. **8**.
7. Jin, W., et al. *Binary Function Clustering Using Semantic Hashes*. in *International Conference on Machine Learning and Applications*. 2012.
8. Bailey, M., et al. *Automated classification and analysis of internet malware*. in *Recent Advances in Intrusion Detection*. 2007.
9. Masud, M.M., L. Khan, and B. Thuraisingham. *A Hybrid Model to Detect Malicious Executables*. in *International Conference on Communications*. 2007.
10. Ghezelbigloo, Z. and M. Vafaeijahan. *Role-opcode vs. opcode: The new method in computer malware detection*. in *International Congress on Technology, Communication and Knowledge*. 2014.
11. Zhang, Y. and B. Wallace, *A Sensitivity Analysis of (and Practitioners' Guide to) Convolutional Neural Networks for Sentence Classification.* Computer Science, 2015.