Long Short-Term Memory based Recurrent Neural Network Architecture to Detect Android Malware using Complex-Flows

Arun Krishnamurthy
Department of Computer Science
University at Buffalo
Buffalo, NY 14214
arunkris@buffalo.edu

Ashwin Vijayakumar
Department of Computer Science
University at Buffalo
Buffalo, NY 14214
ashwinvi@buffalo.edu

Abstract

In this document we take the concept of complex flows introduced in the paper ‘Android Malware Detection using Complex-Flows’ and instead of a support vector machine, apply a deep-learning based classification approach. We explain the architecture of the deep learning model being used and the technical and design decisions that went in.

1 Introduction

1.1 Complex-Flows

To quote the paper this is based on, Complex Flows is based on a mechanism that captures the usage of sensitive mobile resources, but also reveals the structure of this usage as well as the relation between different uses. Using this information, along with a dataset of known malicious and benign apps, we can build a classifier that is able to predict if a new app is malicious or benign. [1]

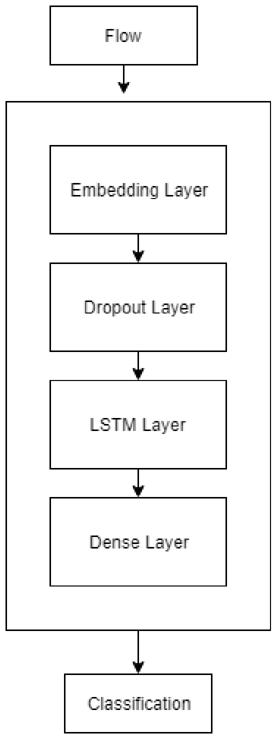
1.2 Recurrent Neural Networks and Long Short-Term Memory units

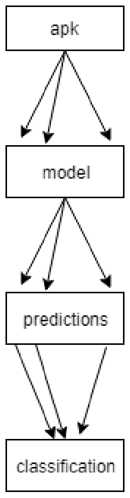
A recurrent neural network (RNN) is a class of artificial neural network where connections between nodes form a directed graph along a sequence. This allows it to exhibit dynamic temporal behavior for a time sequence. Unlike feedforward neural networks, RNNs can use their internal state (memory) to process sequences of inputs. This makes them applicable to tasks such as unsegmented, connected handwriting recognition or speech recognition. [2]

But there are some problems with the way RNNs work mainly the vanishing gradient problem which prevents it from being accurate. In a nutshell, the problem comes from the fact that at each time step during training we are using the same weights to calculate the output. That multiplication is also done during back-propagation. The further we move backwards, the bigger or smaller our error signal becomes. This means that the network experiences difficulty in memorising words from far away in the sequence and makes predictions based on only the most recent ones. [3]

That is why more powerful models like LSTM and GRU come in hand. Solving the above issue, they have become the accepted way of implementing recurrent neural networks. A common LSTM unit is composed of a cell, an input gate, an output gate and a forget gate. The cell is responsible for "remembering" values over arbitrary time intervals; hence the word "memory" in LSTM. Each of the three gates can be thought of as a "conventional" artificial neuron, as in a multi-layer (or feedforward) neural network: that is, they compute an activation (using an activation function) of a weighted sum. Intuitively, they can be thought as regulators of the flow of values that goes through the connections of the LSTM; hence the denotation "gate". There are connections between these gates and the cell. [4]

2 Proposed Architecture





The diagram on the left shows the architecture of the deep learning model that was used. It's just a basic LSTM architecture with an embedding and dropout layer. This classifier learns how to classify a particular flow with a malicious or benign class label.

But the task is to label an entire apk, not individual flows. So, once the model is trained, the inference is not straight-forward. The diagram on the right shows the process for that. While predicting on new apks, we decompose the apk into it's flows first, and then run every flow through the model, and based on which prediction was given the most number of times, that class label is assigned to the entire apk.

3 Computation and Design Constraints

Due to lack of available compute clusters, we had to make some concessions on how much we could train given available compute power. Firstly, we put a 5000 api-call limit to every flow. Basically we truncated out everything the flow has outside this limit. We are intentionally throwing away some information here that the network could potentially learn but we ran into GPU memory issues when we included everything.

We also had to limit the number of apps we can use. We talk about the dataset in a later section, but in terms of constraint, even though we had dataset availability, running everything was infeasible given the timeframe.

Apart from compute, we made some design constraints that we think are actually sound enough to carry forward even when more compute power becomes available. This is to do with the data being considered. There are three ways that one can think of to design an input pipeline to such a training algorithm: the whole apk can be considered as one data point with a class label, each complex-flow can be conisdered as a data point, or, each sequence can be considered as a data point.

Intuitively, it makes more sense to take each apk because we can assign a class label to it that we are confident is correct. We already have a dataset of apks that we know are malicious and benign. But we have wildly varying sizes of api calls, ranging from 1kb to 100mb or more. There are two problems with this approach. First, for the smaller apks, we will end up doing a lot of padding with zeroes and the network spends a lot of time looking at useless information. Secondly, for the really large apps, the network will be looking at hundreds of thousands of api calls per data point, and that means a lot of decimal multiplications. Even LSTMs will not help capture all that information, and the most of that information is wasted and not trained back into the network.

The other option is considering either each complex-flow (essentially a related set of sequences) or each api sequence as a data point. The problem is that each data point needs a label. In the case of malicious apps, we know that the whole app is either malicious based on the dataset, but we can't tell with a degree of certainty if each flow or sequence in that app is malicious. But if we are to do it anyway, it makes sense to label each flow as malicious than each sequence because sequence of calls can be used for many things, and even in a malicious flow there can be benign api sequences. But we think that considering the entire flow as malicious makes more sense as it encapsulates more information than a sequence but since its smaller, it is more tractable than considering an entire apk. It's an acceptable compromise between the three options.

4 Data

We had five different datasets available. On the malicious side, we had the Malgenome dataset which had 1216 apps, two other datasets, Malware_1 (3481 apk) and Malware_2 (3864 apks). On the benign side, we had 1132 Play Store apps from 2014 and 752 Play Store apps from 2016.

As mentioned in a previous section, we were not able to use the entire dataset due to compute constraints. We ended up using 400 apps. But we maintained class balance by taking 200 from malicious side (100 from Malgenome, 50 each from Malware1 and Malware2). On the benign side, we took 100 apps each from Play Store 2014 and Play Store 2016. Having a class balance helps the network learn a representation of both malicious and benign applications and ends up giving us a better classifier.

5 Results

Training was done using data from 400 apps, class balanced equally between benign and malicious. Training ran for 100 epochs, after which we obtained a training accuracy of 90% and a test accuracy of 87%.

While this is an impressive number since the we are using less than 5% of the data, it's important to note that this is the accuracy on flow classification. Which means that for any given flow, the model will give a classification of benign or malicious with 87% accuracy. Each app will have multiple flows so for prediction, as explained in the architecture section, we have to take the classification we get for all flows available for that app and make a final classification decision.

Since we had to manually each apk, and then break it down into flows, predict on every flow, and come up with a label, based on time constraint, we did the prediction only on 20 apps. And we were able to make the correct prediction on 16 apps (handpicked apps that have around 5-10 flows each). Considering that training done just on 400 apps, the results are pretty good.

References

[1] Feng Shen, Justin Del Vecchio, Aziz Mohaisen, Steven Y. Ko, and Lukasz Ziarek. "Android Malware Detection using Complex-Flows"

[2] Recurrent Neural Network. URL: https://en.wikipedia.org/wiki/Recurrent_neural_network

[3] Simeon Kostadinov. How Recurrent Neural Networks work. URL: https://towardsdatascience.com/learn-how-recurrent-neural-networks-work-84e975feaaf7

[4] Long short-term memory. URL: https://en.wikipedia.org/wiki/Long_short-term_memory