

Predicting App Intrusiveness Using LSTM Networks to Analyze App Descriptions

by

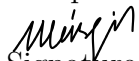
Fernando Montenegro

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Master of Science
(Computer Science and Information Systems)
in The University of Michigan - Flint
2021

Committee:

Advisor:

Halil Bisgin, Ph.D.
Assistant Professor
Computer Science


Signature

April 29, 2021
Date

Reader:

Fadi Mohsen, Ph.D.
University of Groningen
The Netherlands

Signature 

Date 4/29/2021

Reader:

Suleyman Uludag, Ph.D.
Associate Professor
Computer Science

Signature

Date


Apr 29, 2021



COMPUTER SCIENCE AND
INFORMATION SYSTEMS

Copyright © 2021 Fernando Montenegro
All Rights Reserved

To My Family

ACKNOWLEDGEMENTS

nos esse quasi nanos gigantium
humeris incidentes

Bernard of Chartres

I would like to thank everyone whose work made it possible for me to take this next, albeit small, step. Halil Bisgin, Ph.D., Fadi Mohsen, Ph.D., and Nikita Sobers, MS collected the data, explored the dataset and started work using neural networks to make predictions.

My family, Sue, Sophia, and Ricardo, who supported me on long days and nights running computations or writing. And for regularly asking me if I was done.

TABLE OF CONTENTS

DEDICATION	ii
ACKNOWLEDGEMENTS	iii
LIST OF FIGURES	vi
LIST OF ABBREVIATIONS	viii
ABSTRACT	ix
CHAPTER	
I. Android Permissions and Privacy	3
1.0.1 Android Permissions	3
1.0.2 Privacy Score	5
II. Long Short-Term Memory	8
2.0.1 Artificial Neural Networks Artificial Neural Network (ANN)	8
2.0.2 Recurrent Neural Networks Recurrent Neural Net- work (RNN)	10
2.0.3 Long Short-Term Memory (LSTM) Network Archi- ture	12
2.0.4 Word Embeddings	16
2.0.5 Implementing LSTM networks in Keras with Tensor- Flow	17
III. Methodology	19
3.0.1 App Data Preprocessing	20
3.0.2 Network Topology	25

IV. Results	29
V. Related Work	35
VI. Future Work	37
APPENDICES	39
BIBLIOGRAPHY	55

LIST OF FIGURES

Figure

1.1	Distribution of Downloads. Same as the 2005 Pew Research report, our data corpus showed that most Android apps in the store have been downloaded a few thousand times, and only a small percentage have millions of downloads.	4
2.1	A historic timeline of ANNs. Starting in the early days of modern computing until the introduction of LSTM networks.	8
2.2	Two neural networks are shown. The image on the right represents a feed-forward neural network, and the one on the left is a recursive neural network (RNN). Image from <i>IBM</i> (2021), <i>What are recurrent neural networks?</i>	10
2.3	Architecture of the memory cell, from <i>Hochreiter and Schmidhuber</i> (1997). The Constant Error Carousel (CEC) is at the center, with its feedback loop, and the gate units that open and close access to the CEC.	13
3.1	The distribution of raw intrusiveness score.	24
3.2	Distribution of word count in descriptions. The majority of apps have short descriptions, but some apps have more than one thousand word descriptions.	26
3.3	LSTM and Multi Layer Perceptron (MLP) network configurations: a) The input is shown at the top, entering the embeddings layer, and then flowing into the stacked LSTM layers, b) The input is shown at the top, entering the first of two dense layers, with 16 units each. . .	27
3.4	Complete network architecture. Two branches are running in parallel. Two dense layers at the bottom process the combined output to produce a final prediction.	28

4.1	<i>TensorBoard</i> display of the weights in an embeddings layer after a sample run. The two large clumps represent the two values extracted by the network after training.	30
4.2	Screenshot of a <i>TensorBoard</i> network graph display. This graph confirms that the network used by <i>TensorFlow</i> matches the intended network configuration entered in code.	31
4.3	<i>TensorBoard</i> screen showing the training and validation outputs for two sample runs. One run was terminated early due to signs of overfitting.	32
4.4	Epoch accuracy showing the accuracy of a sample training run for our neural network. Validation accuracy plateaus at 73-4%	33
4.5	Epoch loss of the sample training from figure 4.4	33

LIST OF ABBREVIATIONS

ANN Artificial Neural Network

API Application Programming Interface

APK Android Package Kit

CEC Constant Error Carousel

GPS Global Positioning System

GPU Graphics (or recently, General) Processing Unit

LSTM Long Short-Term Memory

MLP Multi Layer Perceptron

NLTK Natural Language Toolkit

RNN Recurrent Neural Network

SMS Short Message Service

ABSTRACT

Predicting App Intrusiveness Using LSTM Networks to Analyze App Descriptions

by

Fernando Montenegro

Mobile apps are at the center of everyone’s daily lives and users give them access to their intimate personal data. Some apps collect more information than they need to perform their job. These apps are called intrusive, and can represent not only a privacy issue, but a security problem for their users. Therefore, it is important to develop methods for figuring out how much an app can detect and collect from its users, and whether that access is in line with their privacy expectations. Several methods have been devised to determine app intrusiveness, a measure that represents how much an app’s data collection deviates from its basic needs. This number, called intrusiveness or privacy score, can guide a user in the process of identifying apps that gather too much personal information. Some of the methods to calculate intrusiveness include analysis of app descriptions and conformity with their programmed behavior. However, most of the existing approaches depend on static analysis that is quite challenging and mandates access to the binaries or source code. This thesis proposes a novel method to determine whether an app is intrusive based on its description, which can allow users to make decisions before downloading. More specifically, we

used a Long Short-Term Memory (LSTM) network to analyze the descriptions, along with a Multi-Layer Perceptron (MLP) network to process metadata provided by other app features. The results show that this combined network structure achieved 79% accuracy in training and 74% accuracy for validation, with 840,000 samples and a 75/25 split between training and validation. Our findings indicate that not only it is possible to use the description and other information available from the app store to predict the intrusiveness of an app, but also that the network required to do the job is fairly small.

Introduction

Mobile apps are at the center of people’s everyday lives, which gives them, and the devices in which they run, unprecedented access to every detail of their activities. Depending on the permissions they have, apps have access to a user’s daily routine, the places she visits and how often she does. They can also access her contacts, have knowledge of how often and how she communicates with them; some apps record and analyze all the sounds around their users, and they can record details about their physical activity. Given this intimate access to users’ personal data, it is important to develop methods for figuring out how much an app can detect from a user, and whether that access is in line with the user’s expectations. The Google Android app store displays the level of access, or permissions, an app is going to have once installed. This is a good starting point, but it may not provide the complete picture.

There have been multiple efforts to provide additional information to users about the intrusiveness of an app. Intrusiveness is a measure of the amount of personal information collected by an app that is not essential to perform its core functions. Some studies have compared the declared permissions to those actually seen upon installation, like *Yu et al. (2016)*; *Qu et al. (2014)*, while other studies have used reverse-engineering to determine app functionality, such as *Taylor and Martinovic (2016)*; *Olukoya et al. (2019)*. This study takes a previously developed intrusiveness score by *Mohsen et al. (2018)* and uses it to train a neural network to predict intrusiveness using only data available in the app store. The goal of the study is to see if is

possible to predict whether an app is intrusive or not by looking at readily available information at the moment of purchase or download. This approach would help any user to make an informed decision on the go, without the need for external tools or databases. The results obtained support the feasibility of this concept, as it will be shown later in this thesis.

According to the *Smartphone Market Share* by IDC (2020), the Android operating system has the largest market share of all mobile operating systems in the world, at an estimated 86.1% in 2020. This study focused on apps in the Google Play store, which is the largest app store in the Android ecosystem, and gives a representative sample of mobile apps. The corpus of data included information readily available to any user before downloading an app, like number of reviews and their average rating, approximate number of downloads, content rating, minimum Android version required, and permissions.

This thesis is organized as follows: Chapter I gives a brief introduction to the Android permissions, how they relate to users' privacy, and discusses the creation of an intrusiveness score. Chapter II provides a brief introduction to the LSTM Networks, their architecture and implementation with TensorFlow. Chapter III talks about the steps followed in the study, from data preprocessing to the architecture of our neural network, followed by the results obtained, in Chapter IV. Related works are referenced in Chapter V, and finally, some possible future work is proposed in Chapter VI.

CHAPTER I

Android Permissions and Privacy

This chapter provides some background information about Android permissions, how they relate to privacy, and how an intrusiveness score can be developed. This score is the starting point for this study, which tried to make a prediction using only information available in the app store.

1.0.1 Android Permissions

In the context of a mobile operating system, permissions refer to the ability to reach certain hardware sensors or services that may be available on any given device. Newer and high-end devices tend to have more sensors of different kinds. The classic examples of sensors are the accelerometer, the front and rear cameras, and the microphone. Examples of services are the Global Positioning System (GPS) location services, phone, Short Message Service (SMS) or texting service, and access to the internet. Researchers like *Enck et al.* (2014) have shown that many apps fail to provide adequate safeguards for their user's information, while *Yu et al.* (2016) also showed that some apps don't even disclose the type of information they are gathering from their users.

In recognition of the potential for privacy violations, mobile operating systems have increased the granularity of the permissions available for users to manage. How-

ever, the average user finds it difficult to understand and manage permissions on their own. There have been multiple efforts, *Quay-de la Vallee et al. (2016)*; *Sarma et al. (2012)*; *Taylor and Martinovic (2016)*, to provide useful information so that users can make informed decisions, but these efforts depend on Google’s or the developers’ ability, or willingness, to provide transparent information about each app’s permissions and behaviors. This dependency creates a large blind spot, which malicious publishers exploit, leaving users at the mercy of Google’s ability to identify bad actors and their apps. Also in this blind spot are legitimate publishers who simply want to capture more information about their users than their competitors, or who unwittingly gather unnecessary data. In addition to all of the above, there could be biases introduced by commercial or other interests that are not aligned with the end-user’s privacy goals.

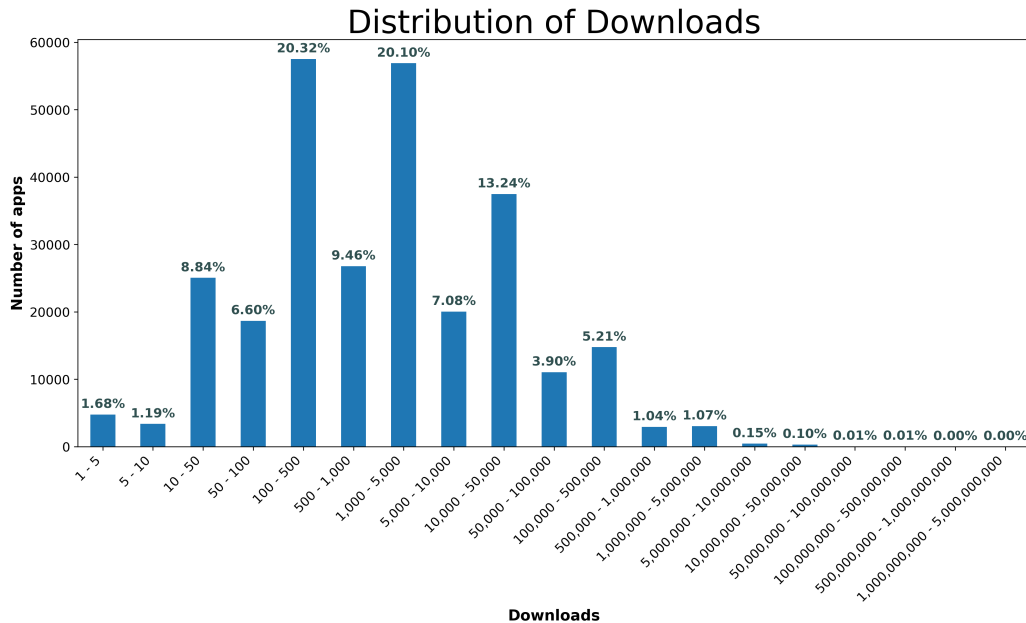


Figure 1.1: Distribution of Downloads. Same as the 2005 Pew Research report, our data corpus showed that most Android apps in the store have been downloaded a few thousand times, and only a small percentage have millions of downloads.

The *Pew Research Center (2015)* published a comprehensive report on Android permissions, their purpose, types, etc. The corpus of data used in this study mirrored their findings. For example, only a small percentage of apps have been downloaded

by millions of users. As shown in Figure 1.1, most apps have been downloaded less than five thousand times. Due to this long tail distribution, all apps with downloads of one hundred thousand or more are grouped into a single category, as discussed in Section 3.0.1.

Similarly, few apps request lots of permissions, and most apps request few permissions, or none at all. According to Pew Research, the average number of permissions requested per app was five, however, the top requested permissions involve access to the internet, access to protected storage, and precise location, all of which can provide sensitive information about a user.

Before the introduction of Android 6.0 *Google* (2015), users had to provide permissions before installing an app, and the permissions were not as granular as they became after this major release. After Android 6.0, users gained the ability to turn individual permissions on or off on a per app basis. As usual, fine-grained access to technical capabilities is a double-edged sword. On one hand, users can have full control of the data each app can access, but on the other hand, many users are overwhelmed by the array of options available to them, as noted by *Alepis and Patsakis* (2019). From this perspective, it is useful to develop a simple measure that can guide end-users through the selection of the apps that best align with their privacy goals. This would be akin to the auto mode option on a sophisticated digital camera, which turns the entire system into a simple point-and-shoot camera, while the user can still set a different mode and change every setting to her needs or desires.

1.0.2 Privacy Score

Most mobile apps present users with a choice, they can choose to have complete privacy, where none of their personal information is accessed and shared, or they can choose usefulness, where an app can access their personal data and use it to provide valuable information. Usually, though, apps do not handle the accessed data locally,

but instead send it to a centralized server to be processed. The server, in turn, returns relevant information for the user to be displayed in their device. Depending on the company’s privacy policy and practices, they can share this information with marketing partners or data brokers, *Leetaru* (2018). The choice ultimately turns into a trade-off between usefulness and privacy.

Allowing an app to access one’s personal information also presents a security risk. As *Chebyshev* (2019) reports in Kaspersky’s *Mobile malware evolution report 2019*, malicious actors continuously try to include their apps in the Google Play store because it is inherently trusted by users. These actors also continuously develop techniques to get around Google’s anti-virus protections, making their malware a real risk, even for security savvy users. And there are also other, more insidious, attack vectors like the use of adware to spread bad software. Legitimate apps that are available for free in the store, are usually supported by ads. An app that requests permissions like location, or the camera, is a prime target for malicious adware. After downloading an infected ad, the app is taken over and all its information becomes available to the attacker. This is another reason why users should install apps that request the strictly minimum number of permissions required to perform their functions.

There have been multiple efforts to help users make informed decisions about the apps they install, *Mohsen et al.* (2018); *Taylor and Martinovic* (2016), in which the authors devise an intrusiveness or privacy score that takes into account the permissions requested by the apps. Using these scores, a user is able to avoid intrusive apps by comparing similar ones and choosing the app with the lowest, or an acceptable score according to their privacy expectations. The studies showed these scores to be useful and effective, but calculating the scores requires access to the apps’ configuration files and is computationally expensive.

This study tried to use only the information available to any user at the moment of browsing, namely the app description and a few other features, to give the user

an option to install an app on a mobile device based on an educated estimate of intrusiveness.

CHAPTER II

Long Short-Term Memory

Long Short-Term Memory (LSTM) networks have revolutionized several fields in artificial intelligence, they have proven to be good at natural language recognition and generation (both spoken and written), as in *Sutskever et al. (2014)*, and they are also good at predicting long series, which can be useful for stock market analysis, music, and many other tasks where data is produced in streams. But before discussing LSTM networks, it is important to show the historical steps that took us to their introduction in 1997. This timeline is represented in figure 2.1.

2.0.1 Artificial Neural Networks ANN

ANNs are closely linked to the early history of modern computing. As the first digital computers emerged, scientists were working together with psychologists to

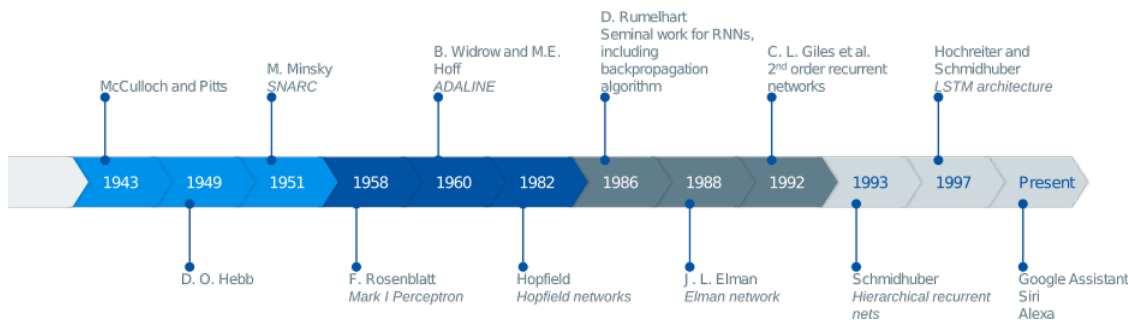


Figure 2.1: A historic timeline of ANNs. Starting in the early days of modern computing until the introduction of LSTM networks.

understand the inner workings of the brain, the ways in which humans learn and remember things, and how we make sense of different concepts. The general idea was to create a machine that could reproduce those processes and do the things humans do with their brains. Neural Networks get their name from their architecture. The basis of the network is a unit that performs a relatively simple function like linear addition, and it is called the neuron. These neurons are interconnected to other neurons with links of adjustable weight. The network learns by adjusting the weights of the connections, which in effect configures, or organizes the network.

In the book by *Plebe and Grasso* (2016), *The brain in silicon*, the authors describe Alan Turing's farsighted *B-type unorganized machine*, which used AND gates for neurons. That paper was not made public, though, until 1996, so they consider the paper by McCulloch and Pitts as an initial point for research on ANNs. Its ideas were soon outdated, but it brought interest to the concept of a neural computer. In their book, *Yadav et al.* (2015), call the 1950s and 1960s *The first golden age of neural networks*, introducing Hebbian networks (1949), which became the model for early implementations like *SNARC* (Stochastic Neural Analog Reinforcement Computer), by Minsky (1951), the *Mark I Perceptron*, built by F. Rosenblatt (1958) for the US Navy, and *ADALINE* (Adaptive Linear Neuron - 1960). *ADALINE* was a successful machine that solved multiple small problems, created by B. Widrow with his graduate students. Ted Hoff was among them, although he is better known today as one of the inventors of the microprocessor. In spite of the early successes, Minsky tried to put a stop to all the research on ANNs in his book *The Perceptron*, declaring the field dead in 1969.

So far, all networks had been feed-forward networks, that is, networks in which the input data follows only one direction on its way to the output. These networks, however, cannot display dynamic behavior, don't have a memory, and are limited to specific applications. Research continued through the 70s and 80s, when *Hopfield*

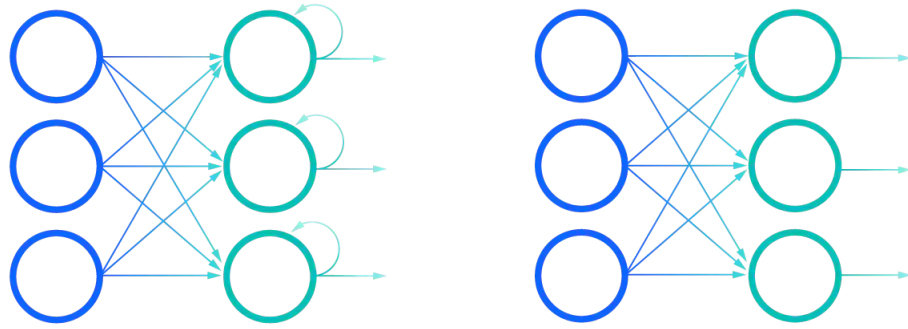


Figure 2.2: Two neural networks are shown. The image on the right represents a feed-forward neural network, and the one on the left is a recursive neural network (RNN). Image from *IBM (2021), What are recurrent neural networks?*

(1982) discovered a type of network that behaved as a content-addressable memory. The Hopfield network was the first popular recurrent neural network (RNN).

2.0.2 Recurrent Neural Networks RNN

These networks emerged as an answer to the need to process inputs that changed over time, and that required some type of short-term memory. The memory function was accomplished by introducing a feedback loop in the cells, which allowed them to act on data from the previous step. (ANNs are discrete systems, that is, they process data in steps, rather than continuously). The RNN architecture required new methods for analysis and training, which were provided by *McClelland et al. (1987)* in their book *Parallel Distributed Processing*. This book, mostly referred in literature as the *Rumelhart* book, generated interest and research on the subject of neural networks. It introduced the *backpropagation* method, used for training RNNs.

RNNs, however, quickly proved difficult to master, or to use for real-world applications. One key issue was their inability to perform on inputs that had 1,000 or more steps, as mentioned by *Hochreiter and Schmidhuber (1997)* in their analysis.

The paper cites multiple solutions that were developed over the years without complete success. Some solutions created other problems, or would simply not work in a real-world scenario, where a response is needed in a reasonable amount of time. In some cases, *Hochreiter and Schmidhuber (1997)* found that the proposed solutions performed worse than simply guessing random weights for the network until a solution was found.

Hochreiter and Schmidhuber (1997) proceed to suggest a new type of cell that they call Long Short-Term Memory (LSTM) in their paper. This new cell solves all the problems present in prior RNNs, and it also performs well in real-world scenarios. Problems with more than 1,000 steps are no problem for this architecture, due to its design because it limits the response to input error, as explained in the next section.

This thesis attempts to provide a tool that can perform natural language analysis in order to extract underlying information that indicates the intrusiveness of a mobile app. This is akin to sentiment analysis, where the networks try to extract the writer's intent on a published text. In human terms, we look at certain keywords that are associated with certain emotions, or certain phrase structures or idioms that are used to provide context in language. RNN, and specifically LSTM networks provide a solution well-suited to the problem at hand in this thesis. In order to analyze the text-based descriptions of Android mobile apps, the neural network must be able to display dynamic behavior and memory. RNNs possess those abilities, as mentioned in *IBM (2021)*. The architecture used must also have a proven record in natural language processing, which eliminates the possibility of getting falsely promising results. *Hochreiter and Schmidhuber (1997)* demonstrated this in the initial paper and it has been further developed and proven by others like *Sutskever et al. (2014)*. This is why LSTM was chosen as the best fit for the problem in this thesis.

2.0.3 LSTM Network Architecture

As mentioned before, Long Short-Term Memory networks were introduced in 1997 by *Hochreiter and Schmidhuber (1997)* as a method to solve the diminishing or exploding gradients problem. When training RNNs with small coefficients, they tend to get smaller and smaller as the error is reduced in search of an optimal solution. Conversely, some gradients could also become very large. The very small and very large values are referred to as diminishing and exploding (respectively) coefficients and they can cause underflow or overflow errors during RNN training. They can also prevent a network from converging because it could get "stuck" on a certain state.

In addition to having issues with error coefficients, RNNs have difficulty processing signals that require long-term memory. Many tasks need both long and short memory capabilities for successful processing. Natural language is one of those tasks where predicting or analyzing the current word may not only need to look at the immediately preceding words, but may also depend on a sentence spoken much earlier. Hochreiter et al. dubbed the cell at the center of their network **long short-term memory** to express its ability to retain information for long periods of time, as well as the short-term retention that RNNs display. We chose to use an LSTM network over other architectures, like RNNs due to their popularity and superior performance in language related tasks.

The LSTM cell can retain memory for extended periods of time by using a central unit with a linear feedback loop called the Constant Error Carousel (CEC). This unit stores a value, and it is flanked by two multiplicative units or gates, in_j and out_j . The first one protects the contents of the CEC from noise or unwanted input from preceding cells, and the latter protects succeeding cells from what could be irrelevant contents in the unit's CEC.

Figure 2.3 shows the architecture of the LSTM as depicted in the original paper. In later works, *Gers et al. (2000)*; *Gers and Schmidhuber (2000)*; *Bayer et al. (2009)*;

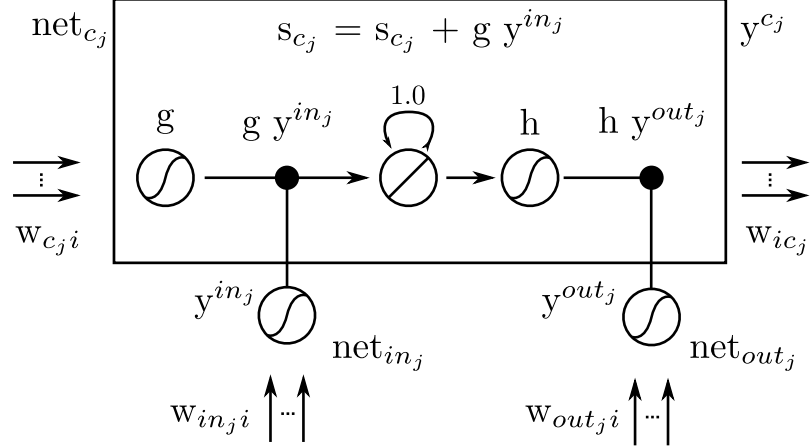


Figure 2.3: Architecture of the memory cell, from *Hochreiter and Schmidhuber (1997)*. The CEC is at the center, with its feedback loop, and the gate units that open and close access to the CEC.

Otte and Zell (2014), the authors have made changes to the original cell, as well as the training algorithms proposed for its use, but *Greff et al. (2017;2015;)* tested and compared multiple architectures in common use and reported that

“none of the variants can improve upon the standard LSTM architecture significantly.”

The only significant improvement made over the original LSTM network is the use of a bidirectional LSTM, which we will describe below.

In the depicted cell we have the activations for out_j and in_j given by

$$y^{out_j}(t) = f_{out_j}(net_{out_j}(t)); y^{in_j} = f_{in_j}(net_{in_j}(t))$$

where net_{out_j} is the input to the multiplicative gate out_j

$$net_{out_j}(t) = \sum_u w_{out_j u} y^u(t-1)$$

for net_{in_j} ,

$$net_{in_j}(t) = \sum_u w_{in_j u} y^u(t-1)$$

and similarly, net_{c_j} is the input to cell c_j , and it is given by

$$net_{c_j}(t) = \sum_u w_{c_j u} y^u(t-1)$$

The indices u can be applied to any type of units present in the network, because their value may be relevant to the current state of the network. Thus, the topology of the network is left to the end user.

The output of the cell is given by

$$y^{c_j}(t) = y^{out_j}(t)h(s_{c_j}(t))$$

where s_{c_j} is the internal state, and it is given by

$$s_{c_j}(0) = 0, s_{c_j}(t) = s_{c_j}(t-1) + y^{in_j}(t)g(net_{c_j}(t)) \text{ for } t > 0$$

The differentiable functions g and h keep the input to and out from the cell within a predetermined set of values.

LSTM cells can be connected with other cells in any configuration desired, however, initial tests only dealt with interconnections in sequences or single layers. Even in their simplest configuration, LSTM networks proved very beneficial from the beginning for some types of analysis. Researchers have also built on the original design and architecture and have created networks using stacked layers of LSTM cells, such as *Sutskever et al.* (2014). Even though it is not clear how this is advantageous from a theoretical perspective, their empirical results demonstrated the power of stacked LSTM layers. Stacked LSTM networks are now commonly used for many applications, and are supported by frameworks like Keras.

As mentioned above, the introduction of bidirectional LSTM networks by *Graves and Schmidhuber* (2005) was the most significant improvement in the design of these

networks. In this type of network, the input is fed forwards and backwards to two separate sequences of LSTM cells connected to the same output. For the particular task of natural language processing, bidirectional LSTM networks display a clear advantage versus other networks, because they can find relationships between the current word or words, and words or phrases spoken earlier that provide context needed for classification or sentiment analysis.

2.0.3.1 Hyperparameters

LSTM networks are more complex than other competing network architectures, which could be considered a disadvantage. It can be difficult to decide how to configure them initially, and how to fine-tune them for the specific task at hand. *Greff et al.* (2017;2015;) analyzed multiple variants of LSTM networks along with their learning rate, hidden layer size, input noise, and momentum. Their paper reports that

“for practical purposes, the hyperparameters can be treated as approximately independent.”

This is a very useful finding for anyone using LSTM networks, because it allows for a much simpler and methodical tuning process for a network. This particular knowledge simplified the process we followed tuning our network, allowing us to change each hyperparameter independently of the others.

In addition to the hyperparameters mentioned above, dropout rate is also important in training. This hyperparameter is interpreted in Keras as the probability that a given sample will be eliminated (dropped out) from the input to the next layer. Intuitively, dropout is useful because it gives neural networks resilience against certain patterns, and allows the networks to correctly classify their input under noisy conditions. RNNs, however, respond poorly to the introduction of dropouts and therefore were missing a valuable tuning tool until *Gal and Ghahramani* (2015) introduced

a technique called variational dropout, in which they removed one symbol out of a group of input sequences, rather than an entire sequence.

To illustrate variational dropout, consider a series of sentences like ‘*The apple tree is tall*’, ‘*That is an old tree*’, and ‘*The forest is full of trees*’. In a traditional dropout scenario, we would remove one of the sentences and use the other two for this particular batch. In variational dropout, we remove the word ‘tree’ or ‘trees’ from all three sentences, and use the resulting sequences for this batch of training. Intuitively, the result is the same as dropout for non-recurrent networks: removing a symbol from any given sequence makes our network more resilient to noise and improves its performance. Variational dropout is an important tool when training RNNs, including LSTMs. Keras supports variational dropout for LSTM layers, although it is called *recurrent_dropout*. We found that this hyperparameter was useful in reducing overfitting for our network.

2.0.4 Word Embeddings

One very popular concept in natural language processing is word embeddings, *Sahlgren* (2015). At a high level, it is the processing of a corpus of text to produce a vectorized representation of each unique word in the corpus. The vector representing each word is comprised of real numbers, and the number of dimensions depends on the type of analysis being performed. A higher number of dimensions allows for a richer, more nuanced representation of each word, which might be required for some tasks.

The concept of representing words with vectors of numbers, or logic relationships has deep roots in linguistics. It is also a key concept used in computer science for information retrieval systems, and has been used since the 60’s. The very specific idea of using neural networks for developing embeddings and using them for language processing that we discuss here is following on the steps of the work by *Bengio et al.*

(2006).

One advantage of word embeddings over, say one-hot encoding, is that the resulting vector represents that word's relationship to other words in the embedded space, *Chollet (2018;2017;)*. With one-hot encoding, all words are equidistant from each other, and it is not possible to infer any relationship between them. On the other hand, word embeddings allow us to perform calculations and determine how closely related any two words are in the given space. Using these calculations, we can derive relationships as in 'woman is to man like mother is to father', or infer that both apple and orange are fruits.

Word embeddings are not without issues. In natural language, we often encounter that words can have different meanings depending on context. 'Orange', for example, can be a fruit, the name of a company, or a color. Traditional word embeddings only assign one vector to each word, which means these three different meanings would be conflated and the meaning of the word could be lost in a particular sequence. Another serious problem with word embeddings is that they tend to reflect our biases, since they learn their relationships from large corpora of text that may have issues. *Bolukbasi et al. (2016)* illustrate this problem in commonly used public word embeddings and have worked on providing guidance on how to remove unwanted bias from the embeddings to improve their accuracy.

2.0.5 Implementing LSTM networks in Keras with TensorFlow

We used the popular framework Keras, developed by *Chollet et al. (2015)*, to implement our neural networks and process the input. Keras provides a high-level, simple to use Application Programming Interface (API) that simplifies many tasks related to neural networks. It is possible to build a stacked LSTM network with a few lines of code. Most of the default values are good for an initial setup, and there is also the ability to go as deep into the details as needed for any specific task.

Also important for us, was the support for Graphics (or recently, General) Processing Units (GPUs), which can provide a significant speed boost to some calculations.

Keras runs on top of other frameworks that provide lower level functionality. We chose TensorFlow, *Abadi et al. (2015)*, which also has a very powerful API for neural network processing. Since we wanted GPU support for our research, we used a specific version of TensorFlow preconfigured for this purpose by *NVIDIA (2020)*. This version of TensorFlow runs inside a Docker container that has all the software needed to run, essentially providing a zero-configuration environment for our project.

The LSTM models that provide full GPU acceleration have some specific restrictions, due to the way the calculations are done. One key item that turned out to be important for our study was the use of variational dropout, which is not supported by the fully optimized LSTM model. We modified other hyperparameters in our network using a setting of zero (or full GPU acceleration) until we could not obtain more improvements, and then started using higher variational dropout values until we reached our final result.

CHAPTER III

Methodology

As mentioned in Section 1.0.1, there have been multiple efforts to assess how intrusive an app is by looking at its permissions and how they relate to the permissions of other similar apps. This study, however, aimed at predicting the intrusiveness of an app based on natural language processing of its description, plus additional hints from its metadata, namely, genre, Android minimum version, download count, content rating, review average, and price. These multiple inputs have very dissimilar forms, which required preprocessing in order to allow neural network processing.

The data corpus for this thesis work was comprised of 1.4M apps gathered from a prior study by *Mohsen et al.* (2018), as previously mentioned. The data included all the features named above plus the package name, which we used to uniquely identify each app. We merged the data with the intrusiveness score, precomputed as a number from 0 to 1, indicating how intrusive an app is. As we'll indicate below, we grouped apps into two groups and labelled them 0 for intrusive, and 1 for non-intrusive apps.

In the dataset obtained, each application has two intrusiveness scores that are calculated based on two different formulas proposed previously by *Mohsen et al.* (2018) and *Taylor and Martinovic* (2016). Both scores rely mainly on the permissions an application has declared in its *AndroidManifest* file and the permissions of its peers. An application that requests common permissions relevant to all applications in the

same genre is considered less intrusive. Similarly, an application that requests unique permissions in comparison to its peers would have a higher intrusiveness score.

3.0.1 App Data Preprocessing

The first set of features in the data corpus are either numeric, or grouped by categories. They were prepared as follows:

- **Genre:** Apps are classified as business, entertainment, etc. When an app had more than one genre associated with it, the first one was used and the others were discarded. The genre listing was converted into a one-hot encoded array number using *scikit-learn's* (*Pedregosa et al. (2011)*) preprocessing API. All zeroes meant no genre was associated with the app.
- **Android minimum version:** Most apps have a required minimum version of the operating system in order to operate correctly (like 2.2 or higher), while some have a range of versions in which they will work (4.1 to 4.3) or allow users to install them. In order to simplify this feature, only the major version of the operating system listed in this field was considered. For example, if an app required Android 2.2 or higher, 2 was used. Similarly, for an app with a range from 4.1 to 4.3, the major version of the lowest OS required was used, in this case 4. Apps listed as 'Varies with Device' had their own category. This category was also encoded as a one-hot array.
- **Download count :** The exact count number is not provided by Google, which provides a category range instead. The range of downloads is very large, from zero to over one billion, but in reality, most apps are actually downloaded a few thousand times. Google's lower categories are kept as separate categories, but all apps with 100,000 downloads or more are grouped into a single category in this study. This grouping left 11 categories, which are also encoded as a one-hot

Feature	Value	Processed Input
pkgname	com.intsig.camscanner	-
Reviews Average	4.5	0.9
Free	True	1
Content Rating	Everyone	[1 0 0 0 0]
Downloads	50,000,000 - 100,000,000	[0 0 0 0 0 0 0 0 0 0 1]
Minimum Android Version	Varies with Device	[1 0 0 0 0]
Genre	Productivity	[0 0 ... 1 ... 0 0]

Table 3.1: Example of preprocessing for an app called *CamScanner*, its features, and the normalized or one-hot encoded values used for input to the neural network. The *Genre* preprocessed input is a one-hot encoded array of 47 bits with 1 in the 14th position.

array.

- **Content rating:** Similar to other features, ratings are encoded as a one-hot array.
- **Review average:** Google provides reviews as a group of measures, indicating not only the average, but also the number of reviews received and the number of each of the star ratings. In order to simplify this measure, we only took the average review and passed it as a normalized number between zero and one to the neural network. The lowest possible rating is one star, which means that apps with an average of zero do not have any ratings.
- **Price:** As the mobile app markets have evolved, the vast majority of apps are free to install and involve some type of in-app purchase. We converted this field into a binary field, with one representing free apps (with or without in-app purchases) and zero representing apps with any cost.

Table 3.1 shows an example of the preprocessing applied to the information of an app called *CamScanner*. The *Value* column shows the raw values as obtained from the app store, and the final column shows the processed values. After preprocessing, the

values are assembled into a concatenated array, which is fed to the MLP branch of the network, described below.

The app description is the most complex of the features we analyzed, and it required the bulk of our attention for preprocessing. Since the descriptions are human-readable, natural language, and almost completely free-text fields, they can have a large amount of variations and content. Many apps have multi-lingual descriptions, emojis, and some contain complex formatting characters. All this creates problems when trying to analyze the content and meaning of that text.

The first step in our process was to remove formatting characters, *Unicode* characters that might cause problems with the scripts, and punctuation. Next, we proceeded to identify all apps that had a description in English. We used the *langid.py* library, created by *Lui and Baldwin* (2012), to identify the description language, which provided good accuracy. We checked the output by randomly sampling the corpus and performing human verification. We discarded the apps with descriptions in languages other than English. We also removed apps that had multi-lingual descriptions, to make sure they did not cause problems later on during our analysis. This step reduced our data sample from 1.4M to about 840k apps.

Once we had a group of English-only apps, we used the Natural Language Toolkit (NLTK), by *Bird et al.* (2009), to remove English language stop words, and to perform stemming. In natural language processing, certain words that are spoken, or used often, but that do not provide significant meaning to a sentence are dubbed stop words. One can usually remove these words during preprocessing to save resources without affecting the final outcome. We used the NLTK stop words list for our preprocessing. Stemming refers to the practice of replacing different word inflections with a base word that conveys the same meaning but can act as their root. As a very simple example, the words *avocado* and *avocados* can be replaced by the word *avocado* and still convey the same meaning in a processed sentence. Stemming has

Step	Output
1	"This app increases your productivity by 10 times!!!\n You will be very happy you used it! :D"
2	This app increases your productivity by times You will be very happy you used it
3	app increases productivity times very happy used
4	app increase product time very happy use
5	[20, 4386, 10267, 2860, 50, 2048, 1873, 0, 0, 0]

Table 3.2: Simplified example of text preprocessing with a 10-word sequence output.

the effect of reducing the size of the vocabulary needed to process any given data corpus.

For the final steps of preprocessing, we used Keras with a TensorFlow back-end. The Keras text preprocessing API has a tokenizer module, which takes the output of the previous steps and generates an index of all the words present in the data. The index is sorted by frequency, with the most common words at the top, and it has a user-selectable vocabulary size. Index 0 is reserved for sequence padding, and index 1 is the out-of-vocabulary symbol. When an app description has fewer words than the maximum number accepted by the neural network, the rest of the positions in the input sequence are padded with zeroes. When the description has words that are not in the vocabulary, they are replaced with the out-of-vocabulary symbol. Table 3.2 illustrates the preprocessing steps taken, from the original app description, to the sequence ready to feed our neural network. In this simplified example of text preprocessing, step 1 shows the original text; step 2 is the text without punctuation, formatting, numbers, or special characters. In step 3 the stop words have been removed. Step 4 shows the effect of stemming. In step 5 we have the final sequence after replacing words with their corresponding index numbers, and we have added padding to form a 10 word sequence.

The vocabulary size must be as large as possible, but it is ultimately restricted by the available resources for the analysis. We did some trial and error tuning for

vocabulary size and settled on 32,768. Since our goal was to predict a score calculated based on Android permissions, we intuitively wanted to make sure our vocabulary included words referring to, or closely related to permission names. For example, one would expect that an app that requires access to any user’s contacts has either the word *contact* in it, or similar words, like *friend*, or *family*. We analyzed the output of the tokenizer and found that the vocabulary included all of the words for the names of the Android permissions we found.

3.0.1.1 Class imbalance

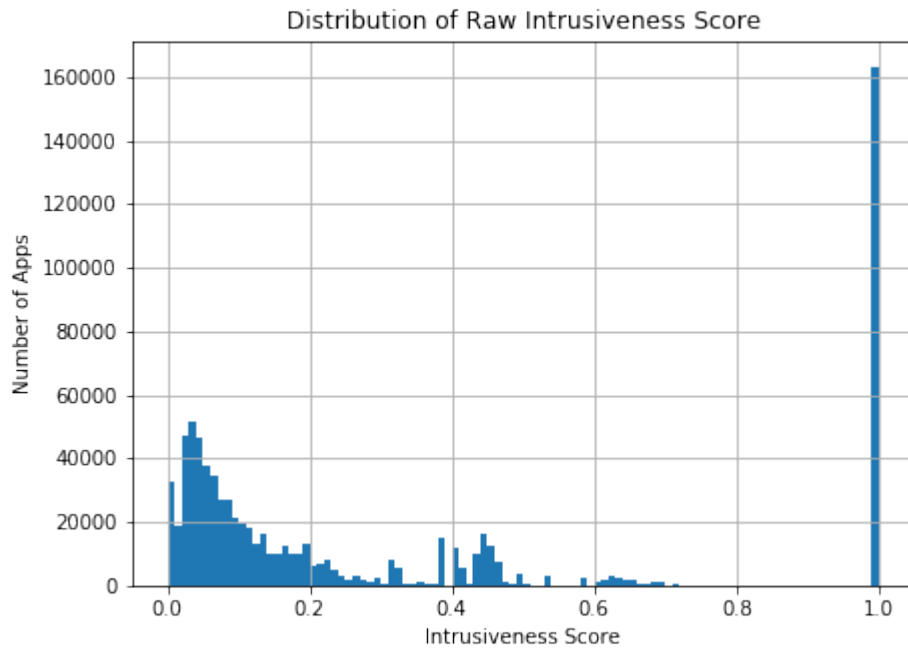


Figure 3.1: The distribution of raw intrusiveness score.

When training neural networks, it is important to maintain class balance, or introduce hyperparameters to guide during training, *Zhi-Hua Zhou and Xu-Ying Liu* (2006). As shown in Figure 3.1, we had many apps with a score of 1.0 compared to lower scores and we wanted to make sure we did not have disproportionately many apps labeled as intrusive or vice versa. Therefore, we arbitrarily chose to use only the top and bottom 35th percentiles of our total data corpus. By doing this, we ensured

that we had the same number of 0 (intrusive) and 1 (non-intrusive) labels in our training and validation samples.

3.0.2 Network Topology

As mentioned above, our analysis included very different types of data, some binary, some categorical, and natural language sequences. Our network design reflected the disparity of inputs and was split into two main branches: the binary and categorical data was handled by an Multi Layer Perceptron (MLP) network, while the language sequences went into a stacked LSTM network with an embeddings input layer. The output of these two branches was fed into two dense layers that performed the final prediction.

3.0.2.1 LSTM branch

The LSTM was the main and largest branch. The main factors contributing to our network size were the maximum length of sequences allowed, the number of dimensions returned by the embedding layer, and the vocabulary size. Some apps have very lengthy descriptions, but they tend to be multi-language descriptions. Since we had only used the first portion of the app descriptions for our language identification preprocessing, we knew that these descriptions started with the English section. We looked at the word count distribution and arbitrarily chose a maximum value of 250 words for our sequences. This worked out to be approximately the 93th percentile of our corpus. Figure 3.2 shows the word count histogram for English-only apps before the final steps of preprocessing. We chose this number as our maximum sequence length.

Google recommends the dimensions returned by the embedding layer be the fourth root of the vocabulary size, *TensorFlow* (2015). However, we have seen multiple implementations using arbitrary numbers higher than 50 in use, and the paper by

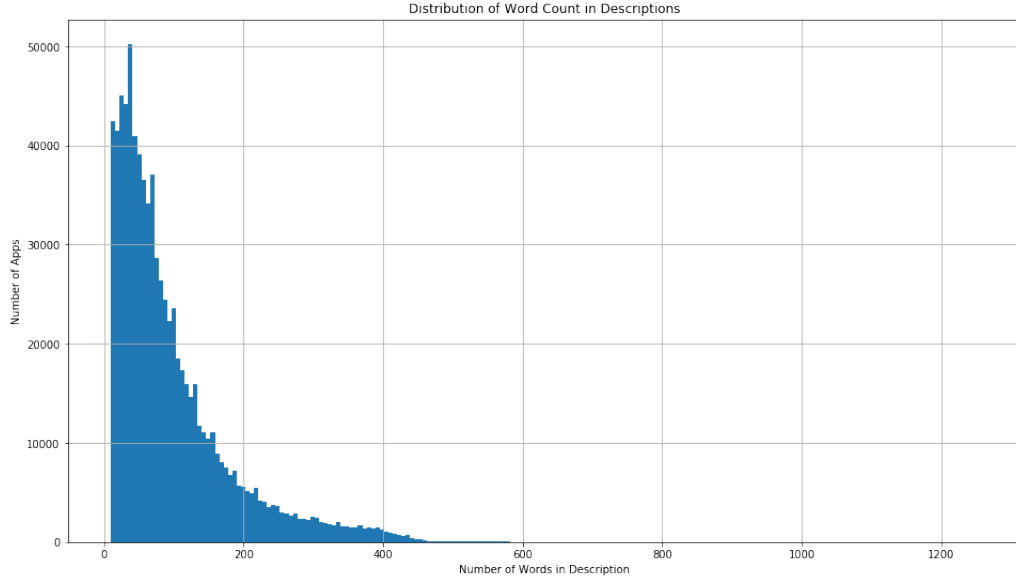


Figure 3.2: Distribution of word count in descriptions. The majority of apps have short descriptions, but some apps have more than one thousand word descriptions.

Seshadhri et al. (2020) suggests that using too few dimensions will cause a model to miss some details that might be important for any given analysis. We started with a value of 16 dimensions, close to the number suggested by Google, and tried multiple values as high as 128, and as low as 4. For our particular problem, 4 dimensions gave the best results, with the added benefit that our computations ran orders of magnitude faster. As we had mentioned, our vocabulary was 32,768 words. Figure 3.0.2.1 shows the final configuration of our LSTM branch.

The output of the embeddings layer was connected to a stacked LSTM formed by three layers. During all our tests we kept the second LSTM layer at one half of the number of units of the top layer, and the next LSTM layer at one quarter of the number of units. For our final configuration, this meant that the bottom two layers had 2 and 1 units respectively. In multiple tests, the stacked LSTM configuration performed better than a single layer, in accordance with initial findings by Sutskever et al. *Sutskever et al. (2014)*. The output of the final LSTM layer went to a dense layer, explained after the MLP branch below.

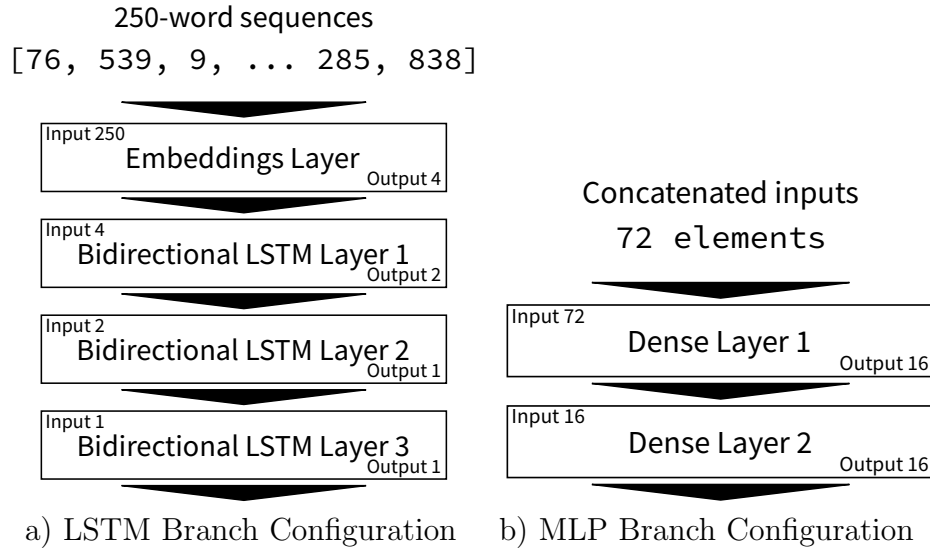


Figure 3.3: LSTM and MLP network configurations: a) The input is shown at the top, entering the embeddings layer, and then flowing into the stacked LSTM layers, b) The input is shown at the top, entering the first of two dense layers, with 16 units each.

3.0.2.2 MLP branch

While the natural language sequences required special treatment with an embeddings layer, the other inputs only required basic normalization before entering our neural network. An MLP network handled all of the other inputs with the topology shown in Figure 3.0.2.1. The input was formed by concatenating all the features, except app description, in an array 72 elements wide. The input went into a dense (fully connected) layer with 16 units. The output of this layer was connected to a second dense layer with 16 units as well. Similar to the testing we performed to find the optimum number of embedding dimensions, we tested multiple sizes of dense layers for our MLP branch. We tried layers as large as 128 units, but we got better results with smaller layers, with a final size of 16. Larger sizes led to overfitting.

3.0.2.3 Combined network output

The output of the LSTM and MLP branches was concatenated and fed into the output layers. The first layer was a dense layer of 4 units, and the final output layer

had one unit with a single output, as shown in figure 3.4.

The entire network was trained with 214,639 samples, which were 75% of our entire corpus, and we used 71,547 samples for validation. Our batch size was 32,768.

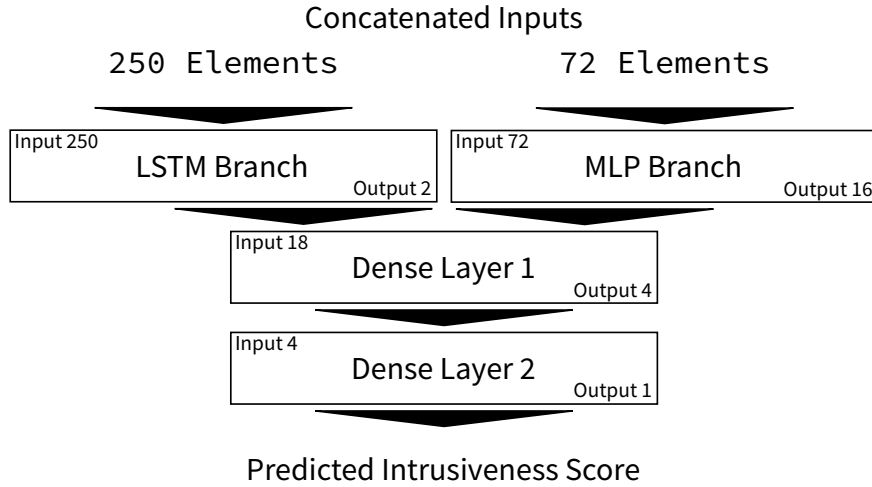


Figure 3.4: Complete network architecture. Two branches are running in parallel. Two dense layers at the bottom process the combined output to produce a final prediction.

CHAPTER IV

Results

We used *TensorBoard* to gather statistics about our training runs. The main advantage of *TensorBoard* is the ability to gather and visualize multiple metrics for a network during and after the different runs. The dashboard allowed us to make adjustments during the initial phases of the study and easily compare the effects. It is also possible to verify the network configuration in a visual manner, which came in handy for our network, given its parallel branches.

Figure 4.1 shows a three-dimensional projection of the embeddings' weights on a sample run of the network. Once the network has been trained, it develops a model of the relationships between the different words in the vocabulary. The constellation of points shows the relative relation of the words in the vocabulary used by the embeddings layer. Intuitively, this graph shows how different words in the vocabulary tend to group in two opposite poles, which for this problem *intrusive* and *non-intrusive*. Some words are neutral to a larger or lesser degree, and they appear somewhere in between the two large groups of words. These visualizations are useful during development and fine-tuning of a network, because they allow to view at a glance how the network is behaving during that snapshot.

Another important feature of *TensorBoard* is the ability to view the network graph to confirm that the network described in code is in fact the network needed for the

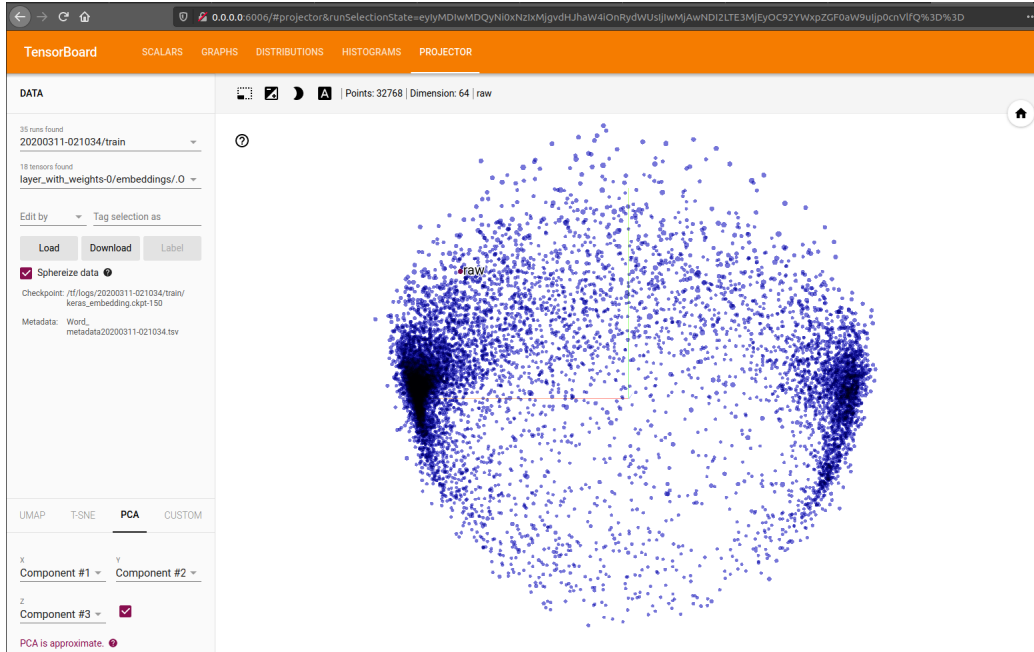


Figure 4.1: *TensorBoard* display of the weights in an embeddings layer after a sample run. The two large clumps represent the two values extracted by the network after training.

analysis. The *Keras* framework simplifies the task of creating and configuring neural networks, but it also creates the possibility that a small bug in the code can change the network architecture in a significant manner. Figure 4.2 shows a sample run of the network. We can visually confirm that there are two parallel branches, one feed-forward branch comprised of two dense (fully connected) layers, and one bi-directional branch. Double-clicking on the latter branch opens up a graph showing the stacked LSTM layers that form it.

Depending on the type of analysis, it may be a good thing to view the neural network progress as it performs multiple epochs, or even after a given number of steps within the epoch. *TensorBoard* allows to do that, plus it allows the user to view multiple runs side-by-side to perform comparisons. Figure 4.3 shows two sample runs in which different hyperparameters have been tweaked. One run (denoted by the orange and grey lines) showed signs of an early convergence, but the validation loss started a significant uptick, which is a sign of overfitting. As a result, the sample run

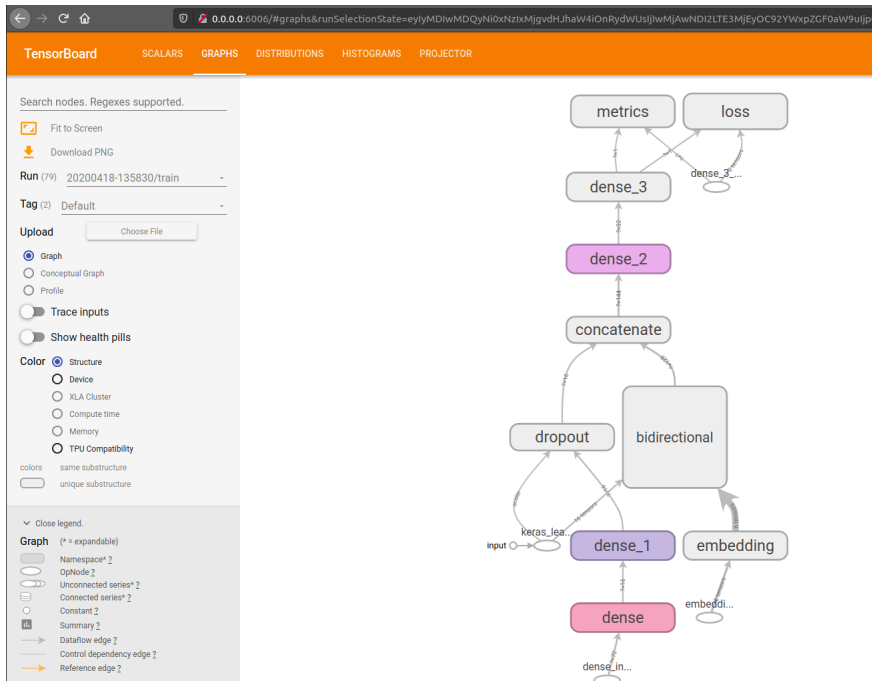


Figure 4.2: Screenshot of a *TensorBoard* network graph display. This graph confirms that the network used by *TensorFlow* matches the intended network configuration entered in code.

was terminated early, saving time and computing resources. The other set of lines (red and blue) represent a slower time to converge, but the network stabilized at a validation rate between 73 and 74% (the best result obtained in this study). When faced with a difficult problem, or a network that needs fine-tuning, *TensorBoard* provides an excellent platform to view and compare multiple results, and potentially generate ideas for improvements.

We tested many configurations for both branches, the LSTM and MLP, and it was very interesting to see that we obtained the best results with the smallest networks we tested. Intuitively, we expected our network to be larger than what we finally used. For example, as mentioned before, *Seshadhri et al. (2020)* showed that it is not a good strategy to use few dimensions for embeddings. In their findings, not using enough dimensions to properly represent the nuances of a given problem results in an inexact model that can produce flawed results. We also expected our LSTM

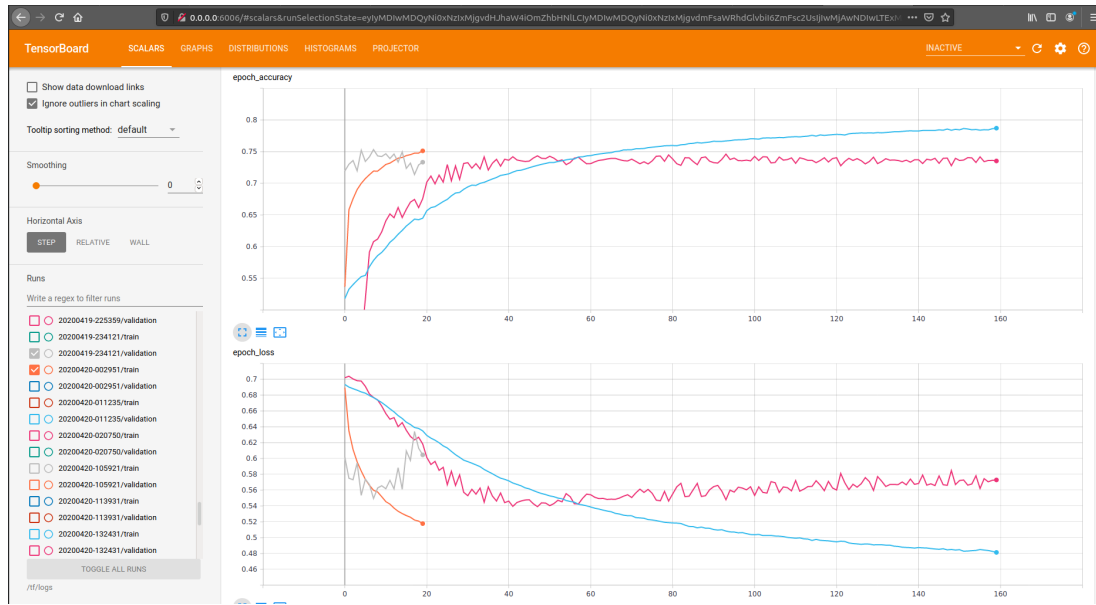


Figure 4.3: *TensorBoard* screen showing the training and validation outputs for two sample runs. One run was terminated early due to signs of overfitting.

network to use many more cells since we were dealing with a fairly large vocabulary, plus sequences of up to 250 words. However, our objective was very specific and used a narrow meaning of words, in a limited environment. These factors may have contributed to make it fairly simple to determine the intrusiveness score for a given app.

As shown in Figure 4.4, the training accuracy (blue line) was about 78% at 160 epochs. And the validation accuracy (red line) was between 73 and 74%, which showed that the neural network was able to correctly predict the intrusiveness on that percentage of tests. The network started stabilizing at around 40 epochs and maintained its accuracy thereafter. Figure 4.5 shows the loss values obtained during the same sample run. From the perspective of an end user, this result means that they can use this network as a way to gauge how intrusive an app is. They can use this information as part of their decision to install the app, or look for alternatives.

Our results show that it is possible to predict an app’s relative intrusiveness by using its natural language description and cues from its other readily available features.

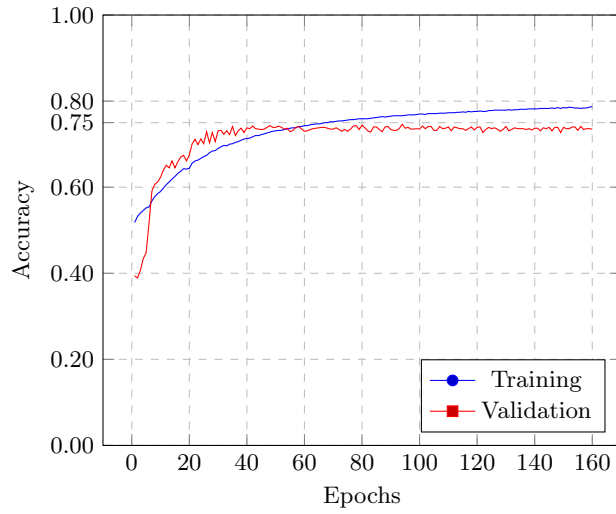


Figure 4.4: Epoch accuracy showing the accuracy of a sample training run for our neural network. Validation accuracy plateaus at 73-4%

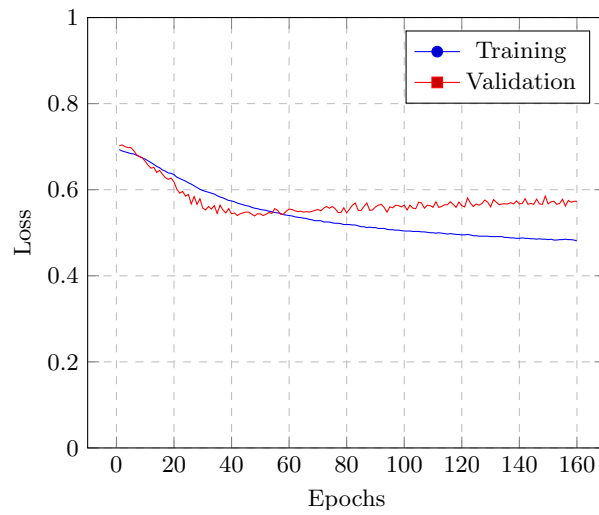


Figure 4.5: Epoch loss of the sample training from figure 4.4

Although more work is needed to improve accuracy, we saw that it is possible to determine, only with the information available at the moment of downloading, whether an app is potentially more intrusive than expected. The user ultimately decides if it is OK to download and install any app, but this approach shows promise as an aid in that choice.

CHAPTER V

Related Work

The relationship between app descriptions and permissions has been an area of interest for other researchers as well. For instance, in their paper, *Qu et al. (2014)* introduce the term *description-to-permission fidelity* to indicate how closely the natural language description of an app matches its permissions. The authors create an automated system to check apps and find that there is, in general, low fidelity in the Google Play store. Even though this paper uses machine learning techniques to extract semantic meaning from app descriptions, its methodology is not nearly as powerful as the use of LSTM networks for the same purpose. Our study also used a much larger data corpus to train and validate the network. We also sought to predict the intrusiveness of an app, rather than look exclusively at its permissions.

Yu et al. (2016) try to improve on prior work that compared an app’s description and its actual permissions in order to identify malicious apps. This new approach uses bytecode and privacy policy analysis to reduce the number of false positives from prior attempts. Although low level analysis of an app can yield very accurate results, it is not feasible to implement for the average end-user. Part of the object of this study was to provide a method that could be used by anyone.

In another study conducted by *Mohsen et al. (2018)*, the authors set the stage for our current study. Namely, the concept of *Intrusiveness Score* is introduced and

calculated for a large set of apps obtained from the Google app store. As mentioned before, our study aimed to predict the intrusiveness of an app using a simplified version of this score, which we used to train and validate our network.

Taylor and Martinovic (2016) propose *Securank* using contextual analysis in their paper, which is based on the idea that similar apps, i.e. apps that try to solve the same problem for their users, should have similar permissions. Consequently, apps requesting more permissions than their peers are arguably more intrusive. Although potentially very useful, SecuRank requires low-level analysis of the apps it ranks and it cannot offer assistance for new apps that have not been analyzed yet. The intrusiveness score developed in *Mohsen et al.* (2018) takes into account the concept of peer applications. Predicting it also helps users achieve their desired level of privacy.

Similar to aforementioned work, *Olukoya et al.* (2019) seek to uncover malicious apps by identifying differences between their natural language description and their actual behavior. Their paper on the work initiated by *Qu et al.* (2014) and others, and they do achieve an improved accuracy. However, their model requires access to the actual app package (Android Package Kit (APK)), which makes it difficult to implement, especially on mobile devices.

CHAPTER VI

Future Work

There are several things that we would like to propose for future work in this area. As there are always new apps and new versions of Android, it would be interesting to perform a similar study with more recent data. In addition, we could also look into obtaining other data points that might give additional insight into any given app's intrusiveness. Two additional data points that come to mind are download size and minimum memory required. One could argue that an app that uses more information after requesting extra permissions, needs to have a larger download size and memory footprint than its non-intrusive counterparts. These two extra data points could help improve the accuracy of prediction.

Another interesting possibility would be the idea of implementing this network in a mobile device. It could analyze prospective apps and help its user make a determination on whether to download and install a new app on their device. The fact that our best performing networks were small, increases the feasibility of this idea. A mobile app could also learn from its user and use that additional knowledge to increase or decrease the level of privacy used for future predictions. Similar to a Netflix algorithm *James Bennet* (2007), it could be possible to offer alternatives to the current app based on other users' similar privacy goals and apps' intrusiveness.

It would also be interesting to try different network architectures, like bidirectional

GRUs, for example.

We could also propose further enhancements to the preprocessing of the Description field. Some ideas would be to replace emojis with words that represent them, run a spell check, try to infer additional meaning from the formatting that has been removed, etc.

APPENDICES

APPENDIX A

Preprocessing code

The code in this section was executed in the context of a Jupyter notebook. By splitting the code into cells, it was easy to debug, or experiment with the input data for the neural network. The code is shown here as a single listing, but it is highly recommended to execute in a similar environment in order to get familiar with the code, or to work on it for future work.

Additionally, the code was executed inside a Docker container, which made it easy to set up the environment needed for the Nvidia GPU, along with the proper drivers and TensorFlow software. One disadvantage of the Docker container was that the image could not be updated select software packages needed for the preprocessing. As a result, the first portion of the code needed to be executed each time the Docker image started. In a regular environment, these statements would be executed only once on the target machine and would persist.

This code is available on GitHub at:

https://github.com/fmontene-umflint/lstm_intrusiveness

```
# Update the VM with the latest and greatest  
!pip install --upgrade pip  
!pip install langid
```

```

!pip install pandas
!pip install nltk

# all the imports
from __future__ import absolute_import, division, print_function, unicode_literals

import langid
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import nltk
from nltk.stem import SnowballStemmer, PorterStemmer
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
import os
import re
import string

# Need to download stop words corpus
nltk.download('stopwords')
# Need to download punkt
nltk.download('punkt')

# Read the files into Pandas dataframes for processing
# data1 and data2 will contain 'pkgname' and 'Descriptions'
data1 = pd.read_csv('ContextualDataDescriptionWhatsNew-part1.csv')
data2 = pd.read_csv('ContextualDataDescriptionWhatsNew-part2.csv')
# data will contain the column 'ID', which is the same as 'pkgname' in data1,
# and other columns called 'Genr', 'Developer',
# plus 4 different permission scores.
# The column called 'permission_2' has the score for our study
data = pd.read_csv('apks_with_scores.csv')

# Temporary dataframes to hold only the columns we will use later on
df1=data1[['pkgname','Description']]
df2=data2[['pkgname','Description']]
df3=data[['ID','permission_2']]

# Perfoming the Merge and getting the class label
# ---

```

```

frames=[df1,df2]
df_cat=pd.concat(frames)

# Remove duplicates
df_cat_final=df_cat.drop_duplicates()

# Then we perform a merge using the 'pkgname' and 'ID' columns
df_cat_merge=pd.merge(df_cat_final,df3,left_on=['pkgname'], right_on=['ID'])

# Remove any rows containing NAs, to avoid processing problems
df_cat_merge = df_cat_merge.dropna()

# Save file for later processing
# First, make sure that all the 'Description' fields contain a string
df_cat_merge['Description'] = df_cat_merge['Description'].map(lambda x: str(x))
df_cat_merge.to_csv('CategoryMergeRaw.txt', sep='\t', index=False)

# We can drop the column 'ID', which was used for the match
# and should be a duplicate of 'pkgname'
df_cat_merge = df_cat_merge.drop(columns=['ID'])

# Dataframe update

# A new column called 'desclang' will contain a 2 letter ISO 639-1 code
# For our tests we are only interested in English, represented by 'en'
# We use langid.py to classify the apps according to the description language
##
# NOTE
##
# At this point we ran into problems with language classification due to app
# descriptions that contained multiple languages.
# We found that truncating the number of words provided to the language
# classifier provided much more accurate classification
trunc_words = 350

df_cat_merge = df_cat_merge.assign(
    desclang=df_cat_merge['Description'].apply(
        lambda row: langid.classify(' '.join(row.split()[:trunc_words]))[0]
    )
)

```

```

# Once again, remove any NAs
df_cat_merge = df_cat_merge.dropna()

# Save file for later processing
df_cat_merge.to_csv('CategoryMergeLanguage2.txt', sep='\t', index=False)

# remove non-English apps
df_cat_merge = df_cat_merge.where(df_cat_merge['desclang'] == 'en')

# Save file for later processing
df_cat_merge.to_csv('CategoryMergeEnglish2.txt', sep='\t', index=False)

# Remove special characters
# NOTE: This will need to be tweaked for any languages other than English
nospace = lambda x: re.sub(
    r'[^a-zA-Z_\s]', # Regular expression removing non-alpha characters
    '',
    re.sub(
        r'[^a-zA-Z_\s]', # Replace non-alpha characters for spaces
        '\s',
        re.sub(
            # Remove tabs, new line, carriage returns, form feeds,
            # and UNICODE characters
            r'(\t|\n|\r|\f|\u[0-9a-fA-F]{4})',
            '',
            str(x)
        )
    )
)

# Remove stop words
sw = set(nltk.corpus.stopwords.words('english'))
nostops = lambda x: '\s'.join(
    str(word).lower() for word in nltk.tokenize.word_tokenize(x) if word not in sw
)

# Putting it all together and removing extra spaces
cleanup = lambda x: re.sub(r'\s+', '\s', nostops(nospace(x)))

# All descriptions
# They should be, but let's make sure all 'Description' fields are strings

```

```

df_cat_merge['Description'] = df_cat_merge['Description'].map(lambda x: str(x))

# Save pre-processed data for later use
df_cat_merge.to_csv('CategoryMergePreProcessed2.txt', sep='\t', index=False)

# Prepare the Snowball Stemmer
stemmer = nltk.stem.SnowballStemmer('english')

df_cat_merge['Description'] = df_cat_merge['Description'].map(
    lambda x: '␣'.join(stemmer.stem(word) for word in x.split())
)

# Save pre-processed data
df_cat_merge.to_csv('CategoryMergePreProcessedStemming2.txt', sep='\t', index=False)

# Take Genre data from original file above
context_data = data[['ID', 'Genr']]
# Make sure all Genr data is a string
context_data['Genr'] = context_data['Genr'].map(lambda x: str(x))
# Remove blanks
context_data = context_data.dropna()

# Remove ampersands from Genres
context_data['Genr'] = context_data['Genr'].map(lambda x: re.sub(r'&', '', x))
# Use stemmer on the Genre field
context_data['Genr'] = context_data['Genr'].map(
    lambda x: '␣'.join(
        stemmer.stem(word) for word in x.split()
    )
)

df = df_cat_merge.merge(right = context_data,
                        left_on = 'pkgname',
                        right_on = 'ID')

# Remove duplicates and NAs
df = df.drop_duplicates()
df = df.dropna()
df = df.drop(columns='ID')
# Save file for later use
df.to_csv('CategoryStemming2Enhanced.txt', sep='\t', index=False)

```


APPENDIX B

LSTM code

This code is available on GitHub at:

https://github.com/fmontene-umflint/lstm_intrusiveness

```
# Prep for the Docker container
# Update all packages
!pip install --upgrade pip
!pip install pandas
!pip install sklearn

# Load extensions and imports
# Load all external packages using pip before executing this in a Docker container
from __future__ import absolute_import, division, print_function, unicode_literals

from datetime import datetime
import numpy as np
import pandas as pd
import pickle
import os
import matplotlib.pyplot as plt
from datetime import datetime
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM, GRU, Dropout, Activation
from tensorflow.keras.layers import Embedding, Bidirectional, concatenate
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.sequence import pad_sequences
```

```

from tensorflow.keras.preprocessing.text import Tokenizer
from sklearn import preprocessing

# Retrieve preprocessed file
df_new = pd.read_csv('Clean-ContextualDataNormalized.csv', sep = '\t')

# Read preprocessed Descriptions data and merge
df_desc = pd.read_csv('CategoryStemming2Enhanced.txt', sep = '\t')

# Merge the two
df = pd.merge(df_new,df_desc,left_on=['pkgname'], right_on=['pkgname'])

# Free memory up and keep only one dataframe
df_new = None
df_desc = None

# Turn Genr into numbers and normalize diving by 50

numGenre = {'educ' : 1,
            'person' : 2,
            'entertain' : 3,
            'lifestyl' : 4,
            'tool' : 5,
            'busi' : 6,
            'puzzl' : 7,
            'arcad' : 8,
            'casual' : 9,
            'music_audio' : 10,
            'book_refer' : 11,
            'travel_local' : 12,
            'photographi' : 13,
            'product' : 14,
            'health_fit' : 15,
            'sport' : 16,
            'action' : 17,
            'news_magazin' : 18,
            'communic' : 19,
            'social' : 20,
            'financ' : 21,
            'simul' : 22,
            'adventur' : 23,

```

```

    'shop' : 24,
    'race' : 25,
    'medic' : 26,
    'map_navig' : 27,
    'video_player_editor' : 28,
    'trivia' : 29,
    'casino' : 30,
    'board' : 31,
    'card' : 32,
    'strategi' : 33,
    'food_drink' : 34,
    'weather' : 35,
    'word' : 36,
    'art_design' : 37,
    'role_play' : 38,
    'librari_demo' : 39,
    'music' : 40,
    'comic' : 41,
    'beauti' : 42,
    'hous_home' : 43,
    'auto_vehicl' : 44,
    'event' : 45,
    'date' : 46,
    'parent' : 47}

```

```
df['n_genre'] = df['Genr'].map(lambda x: numGenre[x]/50)
```

```
# Data sample and hyperparameter settings
```

```
# Make choices for each run
```

```
# Change this to the VM folder where the data is stored
```

```
# and where TensorBoard logs will be created
```

```
data_folder='/tf/'
```

```
# Number of bins for histograms
```

```
bin_num = 20
```

```
# Set timestamp for logs
```

```
log_timestamp = datetime.now().strftime("%Y%m%d-%H%M%S")
```

```
#
```

```
# Tokenization settings
```

```
#
```

```

# If set to True, this parameter reads a tokenizer object from file
use_saved_tokenizer = False
# Name of the file to use to save/retrieve the tokenizer object
tokenizer_filename = 'tokenizer_hybrid'
# The following tokenizer parameters are only considered if
# use_saved_tokenizer is set to False
# Max number of words considered for the NN. This parameter determines the size
# of the embeddings layer, along with the dimensions below
vocab_size = 32768
# According to
# https://developers.googleblog.com/2017/11/introducing-tensorflow-feature-columns.html
# the embedding dimensions should be  $\text{vocab\_size}^{(1/4)}$ ,
# but they say you can use anything. Most experts use a number
# between 50 and 1000. We started with 16 and went to 64.
# 64 provided good results, but showed overfitting
# Trial and error gave us 4 as the best number of dimensions for this problem
embedding_dim = 4

#
# Input sequence settings
#
# Minimum and maximum length of app descriptions fed to the NN
# Too few words may not provide enough information to make a decision
minimum_length = 70
# Notes: looking at the app data, 120 max words covers 43 percentile of apps
#         250 covers 87.5 percentile
max_length = 250

#
# Sampling settings
#
batch_size = 32768
# Percentile to use
# Only samples below this threshold will be used as 0's and
# samples above 1 - threshold will be used as 1's
percentile = 0.35

#
# Hyperparameters
#
# Recurrent Dropout is the Keras equivalent to Variational Dropout

```

```

# Setting this value to > 0 will disable cuDNN architecture
rec_dropout = 0.25 # Only used for LSTM branch
dropout = 0.3 # Only used for MLP branch
# Adam optimizer settings
# Adjust this accordingly to use a customized optimizer
use_custom_adam = False
# Customize learning rate
# Default rate for Adam is 1e-3
# Only used if use_custom_adam is True
custom_adam = Adam(learning_rate=1e-2)
# Number of epochs to run for each test
num_epochs = 160
# Percent of sample data to use for validation
val_ratio = .25
# Number of cells for MLP Dense layers
num_units = 16

# Tokenization and Sequencing
# ---

trunc_type='post'
oov_tok = "<OOV>"

tokenizer = Tokenizer(num_words= vocab_size, oov_token=oov_tok)
# Obtain tokenizer object
if use_saved_tokenizer:
    # Get object from previously saved file
    token_file = open(tokenizer_filename, 'rb')
    tokenizer = pickle.load(token_file)
    token_file.close()
else:
    # Create a new tokenizer
    tokenizer.fit_on_texts(df['Description'])
    # Save the tokenizer to a file for later use
    token_file = open(tokenizer_filename, 'wb')
    pickle.dump(tokenizer, token_file, protocol=pickle.HIGHEST_PROTOCOL)
    token_file.close()

# Before creating sequences, we are removing descriptions that are too short
# and could be causing problems with the NN
df = df.loc[df['Description'].map(

```

```

    lambda x: len(x.split()) >= minimum_length]

# Add Label column, to be used as target for training
df = df.assign(Label = 2)

# Find the percentile values to be used
# By default, this function skips null values
p2_threshold1 = df['permission_2'].quantile(percentile)
p2_threshold2 = df['permission_2'].quantile(1 - percentile)

print('Thresholds to use are ' +
      str(p2_threshold1) +
      ' and ' +
      str(p2_threshold2)
      )

# if the score is below the threshold1, label = 0,
# if the score is equal or higher than threshold2, set the label = 1

#Setting the class label
column_score = 'Label'
df.loc[df['permission_2'] <= p2_threshold1, ['Label']] = 0
df.loc[df['permission_2'] >= p2_threshold2, ['Label']] = 1

# Remove items that we are not going to use
df = df.loc[df['Label'] < 2]
# Drop any NAs
df = df.dropna()
# Reset the index to avoid blanks
df = df.reset_index(drop=True)
# df

# MLP Branch
# This branch analyses the other characteristics of an app
# Rank of Dimensions importance, based on training accuracy:
#
# Genre
# Android Minimum Version
# Download Count
# Age Rating
# Review Average

```

```

# Free <- Probably unimportant because there are so many free apps

# Assign different inputs to different variables and prepare
# category inputs as one-hot matrices
lb = preprocessing.LabelBinarizer()
# RevAvg is a normalized version of the review averages, goes in as-is
x_1 = df[['RevAvg']]
# free is a boolean indicator showing if the app is free, used as-is
x_2 = df[['free']]
# AgeRating needs one-hot encoding
lb.fit(np.array(list(str(x) for x in df.AgeRating)))
x_3 = lb.transform(np.array(list(str(x) for x in df.AgeRating)))
# d_count is a categorized measure of downloads. Needs one-hot encoding
lb.fit(np.array(list(str(x) for x in df.d_count)))
x_4 = lb.transform(np.array(list(str(x) for x in df.d_count)))
# NormVer also category, also one-hot
lb.fit(np.array(list(str(x) for x in df.NormVer)))
x_5 = lb.transform(np.array(list(str(x) for x in df.NormVer)))
# n_genre needs one-hot encoding
lb.fit(np.array(list(str(x) for x in df.n_genre)))
x_6 = lb.transform(np.array(list(str(x) for x in df.n_genre)))
# Putting it all together
x = np.hstack([np.swapaxes(np.vstack([x_1, x_2]),0,1), x_3, x_4, x_5, x_6])

# x = df[['RevAvg', 'free', 'AgeRating', 'd_count', 'NormVer', 'n_genre']]
y = np.array(list(x for x in df.Label))

mlp = Sequential()
mlp.add(Dense(num_units, activation = 'relu', input_shape = (x.shape[-1],)))
mlp.add(Dense(num_units, activation = 'relu'))
mlp.add(Dropout(dropout))

# LSTM Branch
# This branch does App description analysis

sequences = tokenizer.texts_to_sequences(df['Description'])
data = pad_sequences(sequences,
                    maxlen=max_length,
                    truncating=trunc_type,
                    padding = 'post'
                    )

```

```
# We tokenize the data on the full dataset to make it more robust,  
# especially the word_index
```

```
lstm = Sequential()  
lstm.add(  
    Embedding(  
        vocab_size,  
        embedding_dim,  
        input_length = max_length,  
        mask_zero = True  
    )  
)  
lstm.add(  
    Bidirectional(  
        LSTM(  
            embedding_dim,  
            recurrent_dropout = rec_dropout,  
            dropout = dropout,  
            return_sequences = True  
        )  
    )  
)  
lstm.add(  
    Bidirectional(  
        LSTM(  
            int(embedding_dim/2),  
            recurrent_dropout = rec_dropout,  
            dropout = dropout,  
            return_sequences = True  
        )  
    )  
)  
lstm.add(  
    Bidirectional(  
        LSTM(  
            int(embedding_dim/4),  
            recurrent_dropout = rec_dropout,  
            dropout = dropout  
        )  
    )  
)
```



```

    )

# Merging branches
# Performing regression on the combined output of both branches

# All the metrics to use for our network
metrics = [
    tf.keras.metrics.TruePositives(name='tp'),
    tf.keras.metrics.FalsePositives(name='fp'),
    tf.keras.metrics.TrueNegatives(name='tn'),
    tf.keras.metrics.FalseNegatives(name='fn'),
    tf.keras.metrics.BinaryAccuracy(name='accuracy'),
    tf.keras.metrics.Precision(name='precision'),
    tf.keras.metrics.Recall(name='recall'),
    tf.keras.metrics.AUC(name='auc'),
]

# We create a new Model which will receive its inputs from the
# concatenated outputs of our branches above
merged_inputs = concatenate([mlp.output, lstm.output])

# We don't use the sequential model for this section

merged_output = Dense(4, activation="relu")(merged_inputs)
merged_output = Dense(1, activation="sigmoid")(merged_output)

model = tf.keras.Model(inputs=[mlp.input, lstm.input], outputs=merged_output)

if use_custom_adam:
    model.compile(loss='binary_crossentropy', optimizer=custom_adam, metrics=metrics)
else:
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=metrics)

model.summary()

# Data features
Some histograms to see what our data looks like

df.hist(column='Label', bins=bin_num)
df.hist(column='n_genre', bins=bin_num)

```

```

df.hist (column='NormVer', bins=bin_num)
df.hist (column='d_count', bins=bin_num)
df.hist (column='AgeRating', bins=bin_num)
df.hist (column='free', bins=bin_num)
df.hist (column='RevAvg', bins=bin_num)
df['Label'].value_counts()

logdir = os.path.join(data_folder + "logs", log_timestamp)
print ('Issuing callback for TensorBoard to log directory: ' + logdir)
tb_cbk = tf.keras.callbacks.TensorBoard(log_dir = logdir,
                                         histogram_freq = 1,
                                         update_freq = 'batch',
                                         profile_batch = 0)

## Fit the model
history= model.fit([x, data],
                   y,
                   validation_split = val_ratio,
                   epochs = num_epochs,
                   callbacks = [tb_cbk],
                   verbose = 2,
                   batch_size = batch_size)

def plot_graphs(history, string):
    plt.plot(history.history[string])
    plt.plot(history.history['val_'+string])
    plt.xlabel("Epochs")
    plt.ylabel(string)
    plt.legend([string, 'val_'+string])
    plt.show()

plot_graphs(history, 'loss')

```

BIBLIOGRAPHY

BIBLIOGRAPHY

- Abadi, M., et al. (2015), TensorFlow: Large-scale machine learning on heterogeneous systems, software available from tensorflow.org.
- Alepis, E., and C. Patsakis (2019), Unravelling security issues of runtime permissions in android.
- Bayer, J., D. Wierstra, J. Togelius, and J. Schmidhuber (2009), Evolving memory cell structures for sequence learning, in *Artificial Neural Networks – ICANN 2009*, edited by C. Alippi, M. Polycarpou, C. Panayiotou, and G. Ellinas, pp. 755–764, Springer Berlin Heidelberg, Berlin, Heidelberg.
- Bengio, Y., H. Schwenk, J.-S. Senécal, F. Morin, and J.-L. Gauvain (2006), *Neural Probabilistic Language Models*, vol. 194, pp. 137–186, Springer Berlin Heidelberg, Berlin, Heidelberg.
- Bird, S., E. Klein, and E. Loper (2009), *Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit*, O’Reilly Media, Sebastopol.
- Bolukbasi, T., K.-W. Chang, J. Zou, V. Saligrama, and A. Kalai (2016), Quantifying and reducing stereotypes in word embeddings.
- Chebyshev, V. (2019), Mobile malware evolution 2019, <https://securelist.com/mobile-malware-evolution-2019/96280/>.
- Chollet, F. (2018;2017;), *Deep learning with Python*, 1 ed., Manning Publications Co, Shelter Island, New York.
- Chollet, F., et al. (2015), Keras, <https://keras.io>.
- Enck, W., P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth (2014), Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones, *ACM Transactions on Computer Systems (TOCS)*, 32(2), 1–29.
- Gal, Y., and Z. Ghahramani (2015), A theoretically grounded application of dropout in recurrent neural networks.
- Gers, F. A., and J. Schmidhuber (2000), Recurrent nets that time and count, in *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000.*, vol. 3, pp. 189–194 vol.3, IEEE.

- Gers, F. A., J. Schmidhuber, and F. Cummins (2000), Learning to forget: Continual prediction with lstm, *Neural Computation*, 12(10), 2451–2471.
- Google (2015), Android 6.0 changes, <https://developer.android.com/about/versions/marshmallow/android-6.0-changes>.
- Graves, A., and J. Schmidhuber (2005), Framewise phoneme classification with bidirectional lstm and other neural network architectures, *Neural Networks*, 18(5), 602–610.
- Greff, K., R. K. Srivastava, J. Koutnik, B. R. Steunebrink, and J. Schmidhuber (2017;2015;), Lstm: A search space odyssey, *IEEE Transactions on Neural Networks and Learning Systems*, 28(10), 2222–2232.
- Hochreiter, S., and J. Schmidhuber (1997), Long short-term memory, *Neural Computation*, 9(8), 1735–1780, doi:10.1162/neco.1997.9.8.1735.
- Hopfield, J. J. (1982), Neural networks and physical systems with emergent collective computational abilities, *Proceedings of the National Academy of Sciences - PNAS*, 79(8), 2554–2558.
- IBM (2021), What are recurrent neural networks?, <https://www.ibm.com/cloud/learn/recurrent-neural-networks>.
- IDC (2020), Smartphone market share, <https://www.idc.com/promo/smartphone-market-share/os>.
- James Bennet, S. L. (2007), Proceedings of kdd cup and workshop 2007, http://www.netflixprize.com/assets/NetflixPrizeKDD_to_appear.pdf.
- Leetaru, K. (2018), What does it mean for social media platforms to "sell" our data?, <https://www.forbes.com/sites/kalevleetaru/2018/12/15/what-does-it-mean-for-social-media-platforms-to-sell-our-data/#4e2b10e62d6c>.
- Lui, M., and T. Baldwin (2012), langid.py: An off-the-shelf language identification tool, in *Proceedings of the ACL 2012 System Demonstrations*, pp. 25–30, Association for Computational Linguistics, Jeju Island, Korea.
- McClelland, J. L., D. E. Rumelhart, S. D. P. R. G. University of California, and I. X. O. service) (1987), *Parallel distributed processing: explorations in the microstructure of cognition*, MIT Press, Cambridge, Mass.
- Mohsen, F., H. Abdelhaq, H. Bisgin, A. Jolly, and M. Szczepanski (2018), Countering intrusiveness using new security-centric ranking algorithm built on top of elastic-search, in *17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications / 12th IEEE International Conference On Big Data Science And Engineering, TrustCom/BigDataSE 2018, New York, NY, USA, August 1-3, 2018*, pp. 1048–1057, IEEE, doi:10.1109/TrustCom/BigDataSE.2018.00147.

- NVIDIA (2020), Deep learning frameworks documentation, <https://docs.nvidia.com/deeplearning/frameworks/tensorflow-release-notes/index.html>.
- Olukoya, O., L. Mackenzie, and I. Omoronyia (2019), Security-oriented view of app behaviour using textual descriptions and user-granted permission requests, *Computers & Security*, 89, 101,685, doi:10.1016/j.cose.2019.101685.
- Otte, S., and A. Zell (2014), Dynamic cortex memory: Enhancing recurrent neural networks for gradient-based sequence learning.
- Pedregosa, F., et al. (2011), Scikit-learn: Machine learning in Python, *Journal of Machine Learning Research*, 12, 2825–2830.
- Pew Research Center (2015), Apps permissions in the google play store, <https://www.pewresearch.org/internet/2015/11/10/apps-permissions-in-the-google-play-store/>, washington, D.C., USA.
- Plebe, A., and G. Grasso (2016), The brain in silicon: History, and skepticism, in *History and Philosophy of Computing*, edited by F. Gadducci and M. Tavosanis, pp. 273–286, Springer International Publishing, Cham.
- Qu, Z., V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen (2014), Autocog: Measuring the description-to-permission fidelity in android applications, *Proceedings of the ACM Conference on Computer and Communications Security*, pp. 1354–1365, doi:10.1145/2660267.2660287.
- Quay-de la Vallee, H., P. Selby, and S. Krishnamurthi (2016), On a (per)mission: Building privacy into the app marketplace, in *Proceedings of the 6th Workshop on security and privacy in smartphones and mobile devices*, pp. 63–72, ACM.
- Sahlgren, M. (2015), A brief history of word embeddings, <https://www.linkedin.com/pulse/brief-history-word-embeddings-some-clarifications-magnus-sahlgren/>.
- Sarma, B., N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy (2012), Android permissions: a perspective combining risks and benefits, in *Proceedings of the 17th ACM symposium on access control models and technologies*, pp. 13–22, ACM.
- Seshadhri, C., A. Sharma, A. Stolman, and A. Goel (2020), The impossibility of low-rank representations for triangle-rich complex networks, *Proceedings of the National Academy of Sciences (PNAS)*, 117(11), 5631–5637, doi:10.1073/pnas.1911030117.
- Sutskever, I., O. Vinyals, and Q. V. Le (2014), Sequence to sequence learning with neural networks.

- Taylor, V. F., and I. Martinovic (2016), Securank: Starving permission-hungry apps using contextual permission analysis, in *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '16, pp. 43–52, ACM, New York, NY, USA, doi:10.1145/2994459.2994474.
- TensorFlow (2015), Embeddings, <https://developers.googleblog.com/2017/11/introducing-tensorflow-feature-columns.html>.
- Yadav, N., A. Yadav, and M. Kumar (2015), *History of Neural Networks*, pp. 13–15, Springer Netherlands, Dordrecht, doi:10.1007/978-94-017-9816-7_2.
- Yu, L., X. Luo, C. Qian, and S. Wang (2016), Revisiting the description-to-behavior fidelity in android applications, in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, pp. 415–426.
- Zhi-Hua Zhou, and Xu-Ying Liu (2006), Training cost-sensitive neural networks with methods addressing the class imbalance problem, *IEEE Transactions on Knowledge and Data Engineering*, 18(1), 63–77.