The Application of Deep Neural Networks to Password Validation

Brandon Adler, William Coffey, Adriano DeCastro, Zach Jorgensen
Computer Security Department, Student
Rochester Institute of Technology
Rochester, USA
bca7693@rit.edu, wrc3314@rit.edu, axd4512@g.rit.edu, zcj2515@g.rit.edu

***Abstract*—Weak passwords continue to be a major cause of breaches. The effectiveness of password policies is easily nullified by predictable password schemes such as 'Fall-2019'. In this paper, we present a machine learning model for ranking the strength of passwords. The goal of our approach is to train a model to pick up on common patterns in weak passwords, regardless of whether or not they would be policy compliant. The approach uses a deep neural network to to rank the strength of input passwords from zero to four. Once trained, the model can be used to classify a single password at a time with little computational expense. Our evaluations showed an accuracy of 96 percent with a loss of 0.109.**

I. Introduction

Password strength is a security issue that has plagued industry for almost as long as computing systems have existed. Verizon's 2018 Data Breach Investigations Report indicated that as many as 81% of industry breaches involved the use of weak or compromised passwords [1]. The problem is a difficult one due to the security-convenience trade-off. Standards which require more complex passwords tend to lead to reduced productivity, password reuse, and workarounds that ultimately decrease security. Among these workarounds is the tendency for people to introduce passwords which fulfill the technical requirements put forth by company policy, but miss the spirit of these requirements by adhering to predictable patterns which attackers could still exploit.

Strong password policies serve to prevent two types of attacks: from-scratch brute-forcing and brute-forcing via compromised passwords. In the first form the attacker attempts to uncover weak low-length passwords by attempting every possible character combination. This type of attack is entirely thwarted by standard password requirements. Expanding character sets by requiring special characters and numbers as well as requiring a reasonable minimum length can assure that all passwords would require prohibitive amounts of time to brute-force from nothing. Even if an employee "works around" the password requirements, the act of fulfilling them technically is typically enough to thwart this type of attack.

The more modern and nuanced approach to brute-forcing credentials is to depend on already compromised passwords. Public online repositories exist which contain hundreds of millions of passwords or more. These span a wide array of passwords ranging from short and weak ones acquired via from-scratch brute-forcing to passwords leaked in data-breaches, and passwords acquired from cracking hashes. Rather than trying to create correct passwords from nothing attackers can try preexisting passwords in bulk in hopes of finding the reuse of a common password. In LastPass's third annual global password security report they found that an employee reuses a password an average of 13 times [2]. This means that compromising one account can allow attackers to compromise a variety of other unrelated accounts. There's also the distinct possibility of unintentional password sharing between users where multiple users provide the same password unknowingly. This is an incredibly common scenario on account of the "Birthday Paradox", a statistical quirk which shows that probability of two people sharing a trait raises at an alarming rate as the sample size grows. These types of attacks are unfortunately strengthened by the tendency for employees to develop "workarounds" for policies, as this leads to employees pooling around similar password strategies and increases the chances of them overlapping.

Unfortunately this is not a problem that password requirements can solve. Increasing the intensity of password requirements will only result in new workarounds at best, and a substantial loss of productivity or passwords written out and kept near devices at worst. In this paper we propose an alternative to traditional password requirements: using machine learning to classify passwords and prohibiting passwords which are classified as weak. This solution captures much of the same effect as password policies: a model should identify short, low-entropy passwords as weak and longer or higher-entropy passwords as strong. However this solution is also more capable of nuance. For example, it can reject passwords which use diverse set of characters in a predictable manner or accept passwords which may use a small character set but compensates through additional length.

To this end we developed and trained a password classifier using deep neural network techniques which can ingest passwords and provide them a score to signify their strength. We believe this technique could be employed by registration services in place of, or on top of, traditional password requirements in order to validate that passwords are sufficiently strong to a degree that traditional password requirements cannot.

II. Datasets

One particular challenge we faced for this problem was acquiring datasets to train our model. For bad passwords we used pre-existing bad passwords that are publicly available. In particular we used rockyou.txt, a popular word list containing almost 15 million weak passwords acquired in a breach. All passwords in this source were tagged with a score of 02 and provided to the model for training.

Procuring a dataset for "good" passwords proved to be more of a problem. It's difficult for such a dataset to exist due to the nature of good passwords. They're less likely to be acquired and less likely to be compiled since they're of little use for brute-forcing. Ultimately we came to the conclusion that we would need to generate our own dataset for good passwords.

First, we created a password generator that would pull four random words from a wordlist containing a variety of words of 5 characters or longer. It would put these words together and append them with random characters and numbers to produce a complete password. However we worried that if the good passwords were too homogenized the model would learn to identify the structure of the passwords rather than the more nuanced factors that made them secure. As a result we created a secondary password generator that would produce more erratic passwords. Passwords created with this generator could include between 1 and 3 words, but could also include variable length random character strings in-between or after words. Passwords from this generator would include random character sets with some excluding special or numerical characters. Finally all passwords generated from this generator underwent mangling at the end, replacing random characters with special characters or numbers. Since there are no guarantees that this generator only produced good passwords, passwords generated by this method were scored based on their Shannon entropy value and diversity of character sets. Passwords were discarded below a certain score and were otherwise tagged as a 3 or a 4 based on their score.

By using these two generators, in conjunction, we believe the dataset represented a healthier variety of secure passwords, with the first generator representing more structured higher length passwords and the second generator typically representing higher entropy passwords.

III. Design

A. TensorFlow

TensorFlow is an interface for expressing and implementing machine learning algorithms. The project was started by the Google Brain team in 2011 with the goal of creating a flexible, large scale deep neural network framework. Google officially open-sourced the project in 2015 when they released the TensorFlow API and a reference implementation under the Apache 2.0 license. The end result is a single system that can execute on different hardware ranging from mobile devices to distributed clusters of hundreds of machines. The library allows for easy scaling of training and deployment. With the ease of development and scaling, TensorFlow's abstractions have proved useful for both deep neural networks as well as a variety of other tasks.

TensorFlow's basic model is a directed graph composed of nodes. The graph represents the flow of data in the model. Each node represents a computation. Each node has zero or more inputs and zero or more outputs and is an instance of an operation. Flowing between the nodes are arbitrary dimensioned arrays. TensorFlow also defines control dependencies, or edges along which no data flows. Instead the edges inform the destination node that the source node must finished before the dependence begins execution [3].

Local versus distributed computation is implemented with a master worker relationship, with a client directing the run. If the client, master and all workers are on the same machine, local execution is performed. If this is not the case, distributed execution is performed. Separately, a single worker can have multiple compute devices such as CPU cores or multiple GPU devices. Independently of the local versus distributed determination that TensorFlow makes, the system also manages compute selection for different nodes between different compute devices as well as performing the cross device communication. For our implementation, we decided on local computation with a single GPU compute device. This simplified development and required infrastructure, while also providing sufficient compute for the model to train and execute.

B. Our Network

The input to our deep neural network implementation represents the password that will be input by a user. The input data format we used consists of 70 characters which are formatted to unicode and given to the network as a the integer values of each character. An edge case we handle during this phase is padding the input password if it doesn't meet our network's 70 character requirement. Specifically, we pad the passwords with zeros. We decided to have the network support such a large character input in order to grow for the future and allow for the input of some longer, generated passwords.

The network structure itself we're using consists of 7 dense layers. Dense layers are deeply connected neural network layers that implement an activation function applied to each element going through a node, and are mainly used in simpler neural networks or deep neural networks. The first 6 dense layers of the model contain 500 nodes each and use the *sigmoid* activation function. The activation layer normalizes the output of layer so the input for the next layer contains relatively consistent data. For example, the *sigmoid* function center the output between the values of -1 and 1. This allows the network itself to take a large entropy data source, convert it into more manageable numbers to do mathematical operations on and let the depth of the layers create connections between the normalized data.

The last dense layer is more specific compared to the previous layers since it needs to be a one-hot vector for our structure. A one-hot vector is fantastic for our use case since it only has 5 output nodes and uses the softmax activation

function. The 5 output nodes allow our model to output an integer, 0 through 4, depending on how the input filters through the network which is needed due to our classification scale. The softmax activation function returns a vector of 5 elements, this is always the same as the node size for the layer, and normalizes the 5 elements such that the summation of the elements equals 1. After normalization, the index of the greatest number is the classification prediction given the input data. For example, an output from the softmax function could be '[0.1, 0.2, 0.1, 0.4, 0.2]' and the index '3' has the greatest float value so that is what the model predicts as the data's label.

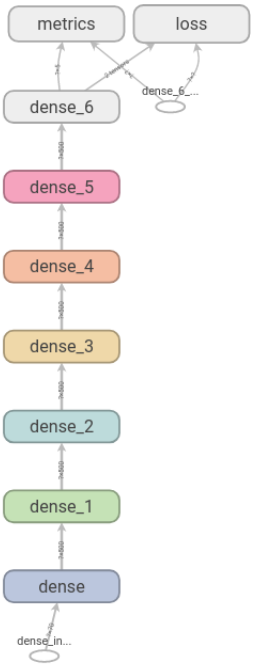

Fig. 1. Model #1 Design

IV. Initial Performance

Our initial model was trained on 10 million passwords randomly chosen from our dataset. Scores were randomly chosen (0-4) and passed to the generator that returned a password for that given score. Batch sizes of 1,000 were used along with 500 steps per Epoch to train the model. The training lasted 20 Epochs resulting in 10 million passwords being ran through the network. After training our model was validated scoring an accuracy of 93% and a loss of 0.178.

Time to train this network was around 23 minutes on a system running a Nvidia GTX 1070 8GB, 20GB RAM and an Intel i5-4690K. This training was performed without multiprocessing and on a single worker due to the nature of Tensorflow multiprocessing on Windows.

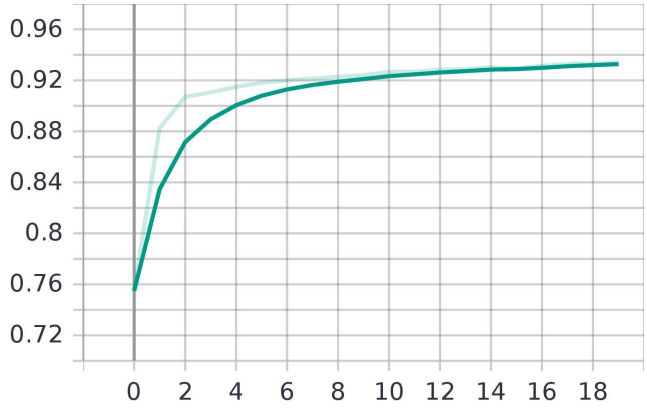

Fig. 2. Model #1 - Epoch Accuracy

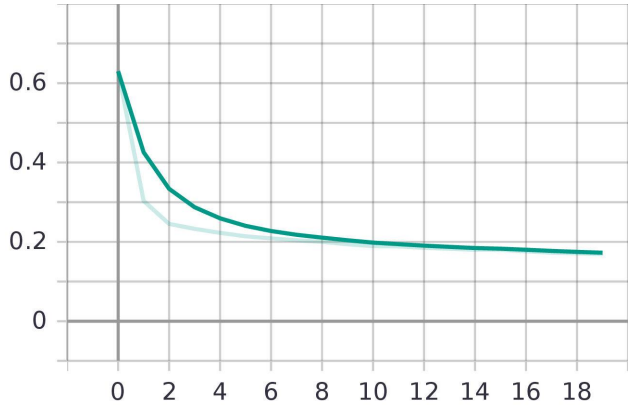

Fig. 3. Model #1 - Epoch Loss

V. Improved Network

The first model produced from our training fulfilled our accuracy requirements during the testing phase but, after some manual testing of passwords, we weren't happy with the overall classifications and felt we could make our testing accuracy better. We eventually decided that the model was overfitting the data and, to combat this, we added a dropout layer to our model's design.

A. Major Changes

The addition of a dropout layer will stop the models tendency to memorize data being passed to it and force it to recognize patterns in the data. The dropout layer was placed in between the forth and fifth dense layers in the network. We assigned an aggressive drop rate of 10% to this layer.

Another major change that increased the accuracy of our second model was to increase the training dataset size and the steps per epoch. The training dataset was increased to 100 million passwords, instead of the initial 10 million passwords, due to the data being generated not meeting our expectations with the generators. This decision, in the end, increased our testing accuracy by another 1% compared to the training we did with only 10 millions passwords for this model.

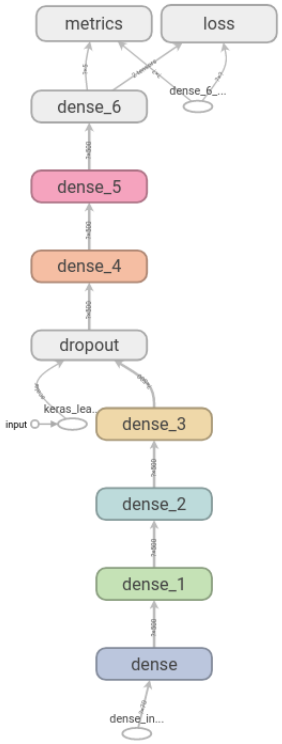

Fig. 4. Model #2 Design

B. Performance

The simple addition of the dropout layer increased the performance of the network significantly. The model scored an accuracy of 96%, a 3% increase over our initial model. The loss of our model after the validation step was 0.109. The batch size used for training was still 1000 passwords, however, the steps per epoch was increased to 5000. This increases the number of batches ran per epoch which is how the model was able to ingest 100 million passwords during the training time. The model was still trained over 20 epochs.

Time to train this network was around 57 minutes on a Nvidia GTX 1050Ti 4GB, 16GB RAM and an Intel i7-8565U. This was performed with 8 workers, multiprocessing enabled, and a max data queue of 20. Enabling multiprocessing significantly increased the speed to train our network.

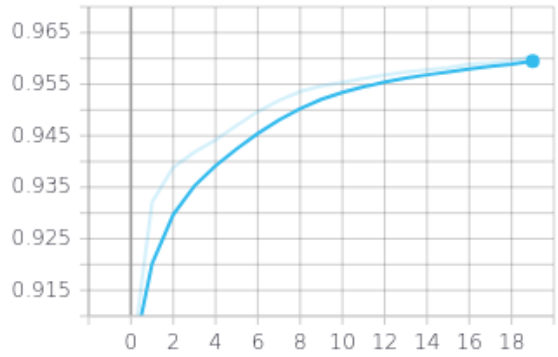

Fig. 5. Model #2 - Epoch Accuracy

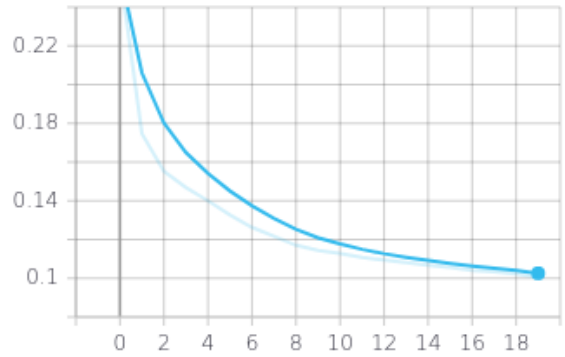

Fig. 6. Model #2 - Epoch Loss

VI. Conclusions & Future Work

There's a growing need for improved identity management. While they have weaknesses that growing method such as multi-factor authentication aim to address, passwords have persisted for this long due to the many beneficial properties they bring to the table. Thus there is a high demand to overcome the flaws they face as an identity management control. While our system needs improvements before it is capable of practical implementation, it has demonstrated the effectiveness of machine learning as a tool for validating passwords.

The most limiting factor of our work was access to datasets and our ability to ingest data. Despite our best efforts to diversify our generated good passwords there were certain traits which our classifier seemed to pick up on which lead to polarized scores in some cases. While a lot of data existed for bad passwords, we lacked the infrastructure to compile and ingest massive amounts data. As a result the limited datasets we chose may have influenced the model's decision making. Future work may strive to improve datasets. The good passwords can be acquired from more diverse generators or by finding a manner to properly acquire good passwords "in-the-wild" in bulk. The bad password dataset could be improved by working on a larger-scale infrastructure that can ingest more data, or alternatively by drawing mixing datasets together to improve diversity rather than pulling entirely from one dataset.

Future work could also opt to look at a more sophisticated model. Our model is far from achieving peak sophistication, and the behavior of different machine learning models when applied to different problems is not always predictable. There may be additional properties which could be applied to our model, or a different model entirely, which would solve this problem with greater efficacy. As such even emulating our work with a different model may serve to further our understanding of how machine learning best tackles this problem.

REFERENCES

[1] Verizon Enterprise Solutions. 2018. 2018 data breach investigations report. (2018). https://enterprise.verizon.com/resources/reports/DBIR_2018_Report.pdf.

[2] Inc LogMeIn. 2019. 3rd annual global password security report. (2019). https://lp-cdn.lastpass.com/lporcamedia/document-library/lastpass/pdf/en/LMI0828a-IAM-LastPass-State-of-the-Password-Report.pdf.

[3] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. Tensorflow: large-scale machine learning on heterogeneous systems. Software available from tensorflow.org. (2015). http://tensorflow.org/.