A Machine Learning Approach to Predicting Passwords

Christoffer Olsen

Kongens Lyngby 2018
IMM-B.Eng-2018

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk IMM-B.Eng-2018

Summary (English)

The goal of the thesis is to investigate whether machine learning models can be used in predicting the sequence of human-created passwords, collected from publicly available database-leaks.

Using a combination of 1-dimensional convolutional layers and dense layers, it is possible to train a machine learning model to give a probabilistic evaluation of password sequences. With this property, it is possible to generate probable passwords along with being able to give a password a strength, based on how likely the machine learning model is to predict the given password.

Passwords generated from the model can be used as dictionary with hashcat, to perform password cracking on hashed passwords. However, the generated passwords are not as efficient at password cracking as popular password dictionaries as `rockyou.txt`, meaning that using machine learning for password prediction still lacks a bit behind when it comes to password cracking.

Summary (Danish)

Målet for denne afhandling er at undersøge hvorvidt maskinlæringsmodeller kan blive brugt til at forudsige sekvensen af menneskeskabte kodeord, indsamlet fra offentlig tilgængelige database-lækager.

Ved brug af en kombination mellem 1-dimensionelle convolutional lag og dense lag, er det muligt at træne en maskinlæringsmodel til at give en sandsynligheds-mæssig evaluering af kodeord. Med denne egenskab er det ligeledes muligt at generere sandsynlige kodeord, baseret på hvor sandsynligt det er at, maskinlæringsmodellen forudsiger det givne kodeord.

Kodeord genereret fra modellen kan blive brugt som en ordbord sammen med værktøjet hashcat, til at udføre 'password cracking' på hashede kodeord. De genererede kodeord er dog ikke lige så effektive til at knække kodeord som nuværende kodeordsordbøger som `rockyou.txt`, betydende at kodeords-forudsigelse ved brug af maskinlæring er lidt bagud, når det kommer til at knække kodeord.

Preface

This thesis was prepared at DTU Compute in fulfillment of the requirements for acquiring a B.Eng in Software Technology.

The thesis deals with predicting passwords using machine learning techniques. The thesis also deals with the implications of being able to predict password sequences and which practical uses it can have.

The thesis consists of background information about password cracking and the different types of methods of how to perform it. The machine learning aspect will be used to predict password sequences and use these for password cracking and other purposes.

Lyngby, 31-January-2018

C. Olsen

Christoffer Olsen

Contents

Chapter 1

Introduction

The purpose of this project is to explore the possibilities of using machine learning to predict passwords. Furthermore, it will be investigated how well a machine learning model compares with today's standard techniques of password guessing, with tools such as hashcat and its dictionary attack method. Lastly the results will be discussed, determining the model's effectiveness and how it might be improved and also which practical implications such a model can have for measuring password strength and password cracking/guessing.

1.1 Hash

A hashing algorithm is a mathematical function that, given a piece of data, attempts to create a unique sequence of bytes, such that no other input of different data will create the same unique sequence of bytes. This enables a way to store passwords so that it is possible to verify the correctness of a password provided by the user, while it is not possible to reverse the algorithm to retrieve the original cleartext passwords.
Example: Given someone's password `password`, when it is hashed via the MD5 hashing function, the resulting MD5 hash, will always be:
'`5f4dcc3b5aa765d61d8327deb882cf99`'.

1.2 Current methods

1.2.1 Password Cracking

1.2.1.1 John the Ripper (JtR)

John the Ripper (JtR) is a software tool that attempts to crack passwords by computing a hash function on a sequence of characters, and then compares whether the output is identical to the hash of the password attempting to be cracked. If they are identical, the passwords should also be identical.

In practice, the tool is optimally used together with a list or dictionary of common passwords, which then computes the hashes and compares the values. Additionally, to improve the the performance, one can choose to apply a 'rule set' to the dictionary, performing changes to the passwords, by for example, appending numbers to end, or replacing e's with 3's (also known as LeetSpeak). These rule sets are built based on common password-choosing behavior obtained from user accounts of leaked databases, from websites such as MySpace and LinkedIn, just to name a few.

With regards to computation, JtR is primarily used with the Central Processing Unit (CPU), which does not scale well when wanting to compute billions of hashes fast and efficiently.

1.2.1.2 Hashcat

Due to the simplicity of computing a hash function, it was attempted to distribute the computation across multiple cores with the help of Graphics Processing Units (GPUs) with their many-core architectures.
Hashcat is a tool which purpose and general idea, is similar to JtR, but utilizing GPUs instead of CPUs for computation of the hash functions. A current generation GPU can compute up to 30 GHashes/s [Gos17]. That is $30,000,000,000$ hashes a second, using the MD5 hashing algorithm.
Because of its computation speed, hashcat is the preferred password cracking tool, if one has the hardware.
Hashcat will be used as the primary password cracking tool, for this project.

1.3 Paper using similar approaches

1.3.1 Fast, Lean and Accurate

Fast, Lean and Accurate [MUS+16], is a research paper targeted towards, making a better and more efficient model for measuring password strengths. The paper explores current methods for password guessing and proposes a new method for password guessing, using Artificial Neural Networks. The neural network turns out to be quite efficient at generating passwords that are being chosen by humans.
As one of the questions they wanted to explore was, if it is possible to represents a password's strength in real-time, in for example, a website. Due to a neural network's high demand for disk space and computation power, it represented some issues achieving this result. With the help of Monte-Carlo search and pre-computing guess-numbers from the neural network output, they were able to measure a password's strength with less than 1 MegaByte of space and a response time below 100 ms. These numbers allowed it to be represented real-time on a website, using JavaScript.

1.4 Password Attributes

There are numerous attributes one could look at to evaluate a password's strength.
One could look at the password's structure such as which characters were used, its ordering and its length.

- **length** - is the number of characters of the password
- **characters sets** - is the different classes of characters included in the password. I.e. `loweralphanum` (Lowercase Alphanumerical), means that the password consists solely of lowercase characters from the English alphabet `[a-z]` including at least one number between `[0-9]`.
- **ordering** - is the structure of the password, which character set is first and which comes after. I.e. `stringdigit` (sequence of characters followed by digits).

Let's look at a few password examples and give them some attributes.

- `password` - length: 8 - character sets: `loweralpha` - ordering: `allstring`.
- `dtu123` - length: 6 - character sets: `loweralphanum` - ordering: `stringdigit`.
- `321PassWord` - length: 11 - character sets: `mixedalphanum` - ordering: `digitstring`.

In general, a password is considered more secure the longer it is, the more different character sets it uses and its ordering is complex. These all increases the number of combinations to try, before an attacker manages to guess the correct one.

To retrieve these statistics given a list of passwords, a tool called `pipal` [Woo14], can be quite useful.

Chapter 2

Data

2.1 Available Data

2.1.1 XSplit

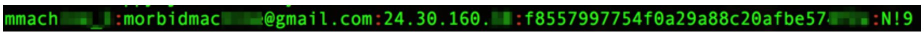
In November 2013 a Livestreaming service called XSplit was compromised, and their entire customer database was leaked, see `https://haveibeenpwned.com/PwnedWebsites#XSplit`.

For this project, $100,000$ random unique passwords from the leak, will be used for measuring and comparing password cracking performance for different attack methods.

2.1.2 OwnedCore.com

In August 2013, a website called OwnedCore.com got its database leaked due to an SQL-injection vulnerability found on their website,
see `https://haveibeenpwned.com/PwnedWebsites#OwnedCore`. The vulnerability allowed an attacker to extract `OwnedCore.com`'s entire user database, revealing about 800,000 entries of usernames, e-mail addresses, IP-addresses and

the user's hashed password. Below is an example of how an entry from the database leak would look like, with a few character blurred-out for anonymity:



```
mmach[redacted]_[redacted]:morbidmac[redacted]@gmail.com:24.30.160.[redacted]:f8557997754f0a29a88c20afbe57[redacted]:N!9
```

The values are colon-separated like in the following format:
<**username**>:<**e-mail**>:<**ip-address**>:<**hashed password**>:<**salt**>

The website is built upon a well-known forum framework called vBulletin. vBulletin hashes each user's password using the following hashing scheme:

```
MD5(MD5(password) + salt)
```

So it first hashes the password with md5, then hashes the md5-hashed password together with a 3-character sequential salt, using md5 again.

When a database is leaked, one of the first things malicious actors do, is attempt to "crack" the passwords. This is usually done utilizing many GPU's and the tools described in Section 1.2.1.2 on page 2. This is done to match up cleartext passwords with usernames and e-mail addresses.

Because people have a tendency to reuse passwords, these database leaks allow attackers to attempt to login with a user's e-mail and password on other websites, such as Facebook or Twitter, obtaining unauthorized access to user accounts of which websites haven't been compromised.

The database leak will be used to train a neural network, to predict password sequences.

Chapter 3

Methods and Implementation

3.1 Performance of different attack-styles using hashcat

There are several methods to cracking passwords, such as bruteforce attacks, dictionary attacks or dictionary attacks with word mutations.

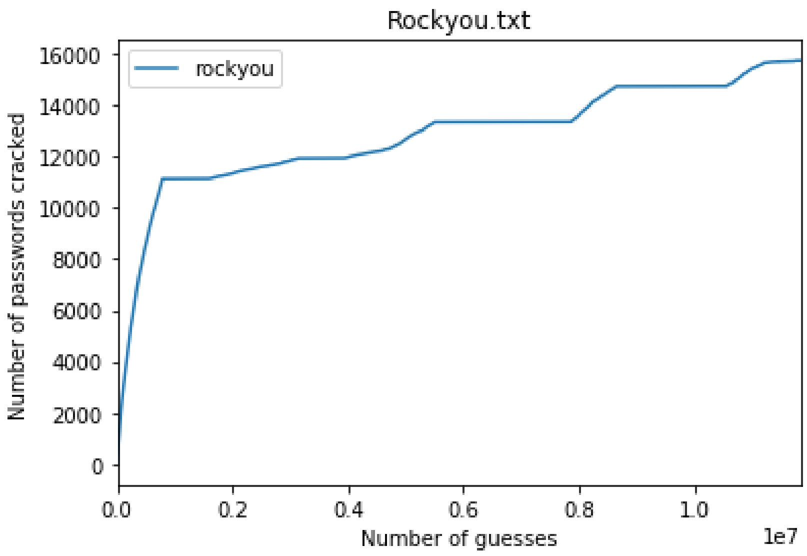
In this section it will be shown how well different methods perform, using hashcat, as in how many guesses it performs, before it cracks a password. The passwords to crack are from a public database-leak, as mentioned in Section 2.1.1 on page 5.

3.1.1 Dictionary attack

A very common and highly successful attack, when it comes to password cracking, is using a list of top-most passwords that have been leaked in the past, and sorted by how often the password was used, putting the most frequent password at the top and the less frequent passwords at the bottom. That is exactly how the dictionary, `rockyou.txt` was made.

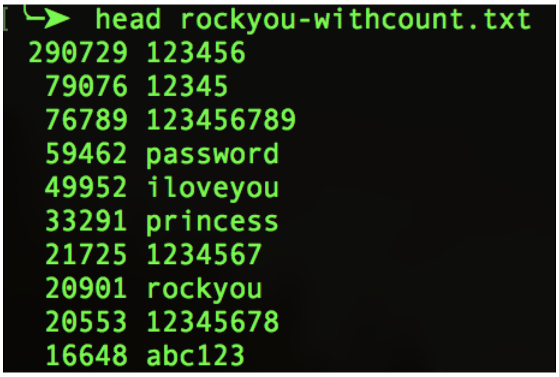
Running hashcat using the dictionary attack method with `rockyou.txt` managed to crack around 15% of the password hashes. Let's look at how many of the passwords were cracked, versus how many guesses were performed.

Figure 3.1: The number of passwords cracked as function of the number of guesses for `rockyou.txt`



From Figure 3.1 it becomes apparent that most of the passwords that were cracked using the `rockyou.txt` dictionary attack, were cracked at the beginning of guesses. This correlates with the fact, that the list had been sorted after how often people uses those passwords.

The top of the `rockyou.txt` list looks like the following, with the number of occurrences on the left, as in what it is sorted by, and the password on the right.



```
[ ╰─➤  head rockyou-withcount.txt
 290729 123456
  79076 12345
  76789 123456789
  59462 password
  49952 iloveyou
  33291 princess
  21725 1234567
  20901 rockyou
  20553 12345678
  16648 abc123
```

Very weak passwords such as only using numbers and passwords with short length, are present at the top.

After the the most probable passwords had been cracked, the graph flattens out a bit more, and for the remaining number of guesses, less passwords were cracked.
After around 14 million guesses, roughly $15,000$ passwords had been cracked. 14 million guesses might sound like a lot, but recall the cracking speed that is achievable as mentioned in Section 1.2.1.2 on page 2, meaning that the passwords were cracked in under a second.

3.1.2 Dictionary + rule set attack

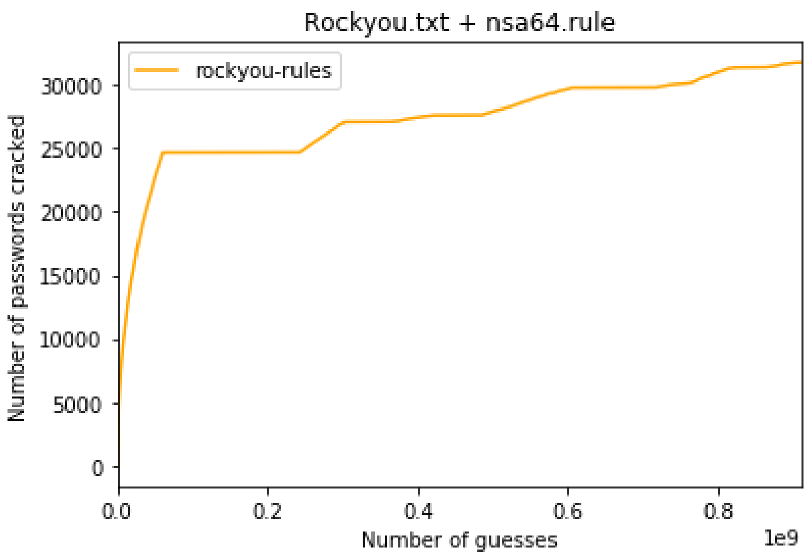
Another very common attack, is using a rule-based approach on top of a dictionary. This method takes each word (password) of the dictionary and mutates the word, to guess passwords that are similar to it.

Consider the base-word `summer`, the rule set applies several mutations to the word such as: `Summer2017`, `Summer1234`, `Summ3r`, just to name a few. In this example, using the `rockyou.txt` dictionary with a relatively basic rule set called `nsa64.rule` [NSA16], around 30% of the password hashes were cracked.

Figure 3.2 on the following page looks very similar to the one with just the dictionary, in terms of shape. However, just about double the amount of hashes were cracked compared to without a rule set. The number of guesses on the other hand had increased to roughly $11,000,000,000$.

This suggests that applying a rule set is usually less efficient, when it comes to minimizing the number of guesses, but with a high cracking speed of up to

Figure 3.2: The number of passwords cracked as function of the number of guesses for `rockyou.txt` + `nsa64.rule`



30 billion a second, shows little to no difference in time, when it comes to cracking the hashes in practice. While the number of guesses had increased almost 1000-fold, the cracking time remained extremely quick.

3.1.3 Very large dictionary attack

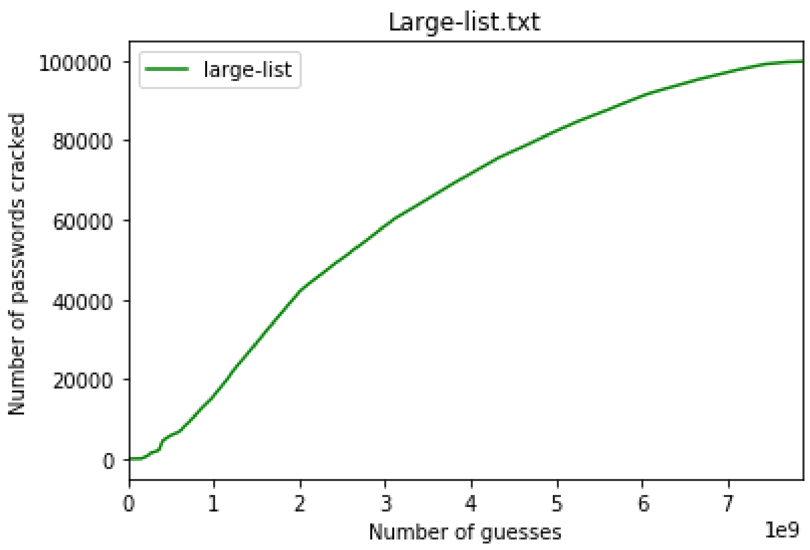
Given the relatively successful dictionary attack with `rockyou.txt`, which is a small list of just over 14 million entries, and the high password cracking, using a huge list would therefore be a great candidate.

`weakpass_2a.txt` is a collection of password from numerous database leaks and holds a whopping 7, 884, 602, 871 passwords [wea]. `weakpass_2a.txt` is sorted alphabetically with the shortest passwords first.

Looking at Figure 3.3 on the next page, it is apparent that `weakpass_2a.txt` is not sorted after most probable passwords, and is more evened out between password cracks and guesses.

This attacks managed to crack almost all 100, 000 passwords, showing the power

Figure 3.3: The number of passwords cracked as function of the number of guesses for a very large dictionary



of knowing many user-chosen passwords and thereby the password-choosing behaviour of humans.

3.1.4 Bruteforce attack

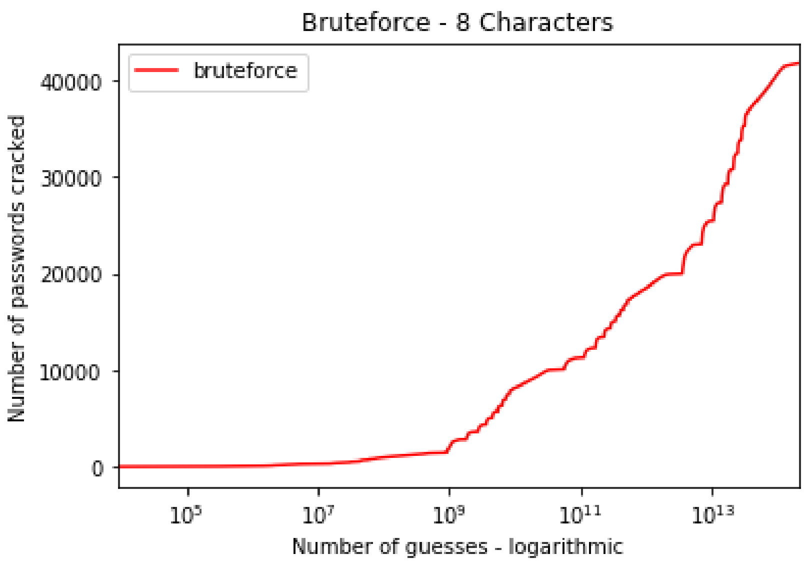
Lastly, the very exhaustive method of a bruteforce attack's performance was tested. The bruteforce attack was initialised to guess every combination of characters, among the set: `[a-z]`,`[A-Z]`,`[0-9]` excluding special characters, between password lengths of 1-8 characters. Some quick calculations reveals that the number of guesses it has to perform is:

$$72^1 + 72^2 + 72^3 + 72^4 + 72^5 + 72^6 + 72^7 + 72^8 \approx 7.324e^{14} \qquad (3.1)$$

This number is substantially larger than those for the other methods, meaning that it will take a while for it to compute all the guesses. For this test, it was computing hashes at a speed of around 47,000 MH/s, resulting in a total running time of just over 4 hours and 20 minutes. In this test, the bruteforce attack managed to crack 41.8% of the hashes.

In Figure 3.4 on the following page it is shown that users, luckily, do not tend to choose passwords with short length, as the majority of hashes are cracked

Figure 3.4: The number of passwords cracked as function of the number of guesses for an 8 character bruteforce attack



at a higher number of guesses, recalling that the method tries short passwords first, and increments the length every time.

As seen in Figure 3.4 bruteforce attacks are very exhaustive and are not very efficient. They require a lot more guesses compared to how many hashes it cracks. It is, however, very thorough.

Bruteforce attacks, tend to lose their effectiveness when users choose passwords greater than 8 characters, as increasing the length of the password by one, increases the number of different combination exponentially. In fact, a bruteforce attack with the character set from the test of a password with length 9, takes:

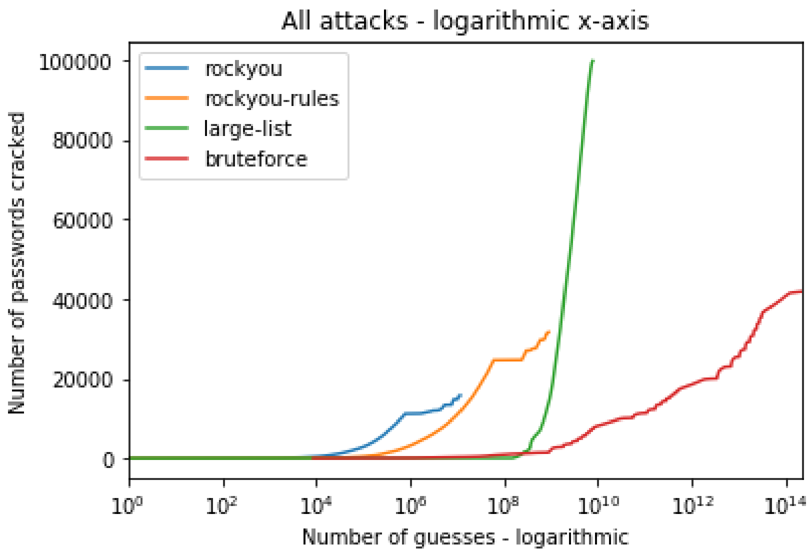
$$\frac{72^9}{47 \cdot 10^9} \approx 1,106,383 \Rightarrow \frac{\frac{1,106,383}{3,600}}{24} \approx 12.8\text{days} \tag{3.2}$$

Which for most purposes is considered, too much. It does however, prove the point that a few extra characters can improve your chance that a password is not cracked if a database, where you have an account, gets hacked or compromised.

3.1.5 Comparing the methods

To sum up the different attack methods' performance, they are compared in a graph in Figure 3.5.

Figure 3.5: The number of passwords cracked as function the number of guesses for all methods



The graph shows that when it comes to cracking hashes, with the fewest number of guesses, a probabilistic approach is most successful, namely the `rockyou.txt` dictionary attack. However, it also does not manage to crack as many as the other methods.

When applying a rule set to the dictionary attack, the number of passwords cracked, is doubled, however, requiring a quite significant extra number of guesses.

The large-list proved to be the most successful of these attacks, managing to crack almost all of the hashes.

Lastly, it is seen that the bruteforce attack, requires an immense amount of guesses compared to how many passwords it managed to crack and is therefore much less efficient.

3.2 Neural Networks for password sequence prediction

For building and training the model, a framework called keras [C+15], was used. Keras provides an easy method of building, training and optimizing machine learning models while also using the tensorflow [AAB+15] backend for cutting-edge performance.

3.2.1 Data representation

The general idea was to build a model, that could predict the following character based on the previous characters. To achieve this, the data was represented as matrices.
Consider the password `oblivion`, the model starts off by having the first character `o` in an n-dimensional vector, depending on the length of the longest password of the training set. It then learns that given the first character, the next character will be `b`. The second time around, it loads in 2 characters, and learns that given the sequence `ob`, the next character is likely to be `l`. This process is repeated until it reaches the password's last character.

So for all passwords in the training set, it learns from them, the probabilities of which character is next, given a sequence of characters.

Programmatically, the password is loaded into n arrays of m length, where n is the length of the longest password in the training set and m is the character set of which is present in the training set, which in this example is `[a-z]`, `[0-9]`. Each index in the arrays of m-length is indicative of a character in the character set. As an example, the character `o` is index number 25. So the array, on the first run-through loads a 1 in index number 25, indicating that, that is the character to predict, based on. On the following iteration, the contents of the previous array is shifted up, to allow for a new character in the front-most position.

3.2.2 Building the model

A few different model implementations were tested, to see which one would bring the best result without over-fitting.
The idea of using convolutional 1-dimensional layers, was proposed due to their property of looking at multiple characters at a time. These convolutional 1d layers turned out to have rather good performance.

Figure 3.6 on the next page shows how the layers are put together and how they are positioned. Every layer is using the relu activation function, except the output-layer, which is using softmax. The model starts off with a convolutional layer with 300 neurons and a filter size of 7. Following that layer, is a dropout layer, with a 20% fraction rate. Then, a dense layer with 300 neurons followed by another convolutional layer, but with 200 neurons and a filter size of 5. It then performs global average pooling followed by a dense layer with 200 neurons into the output layer with 38 neurons, representing each character in character set including a line-terminator.

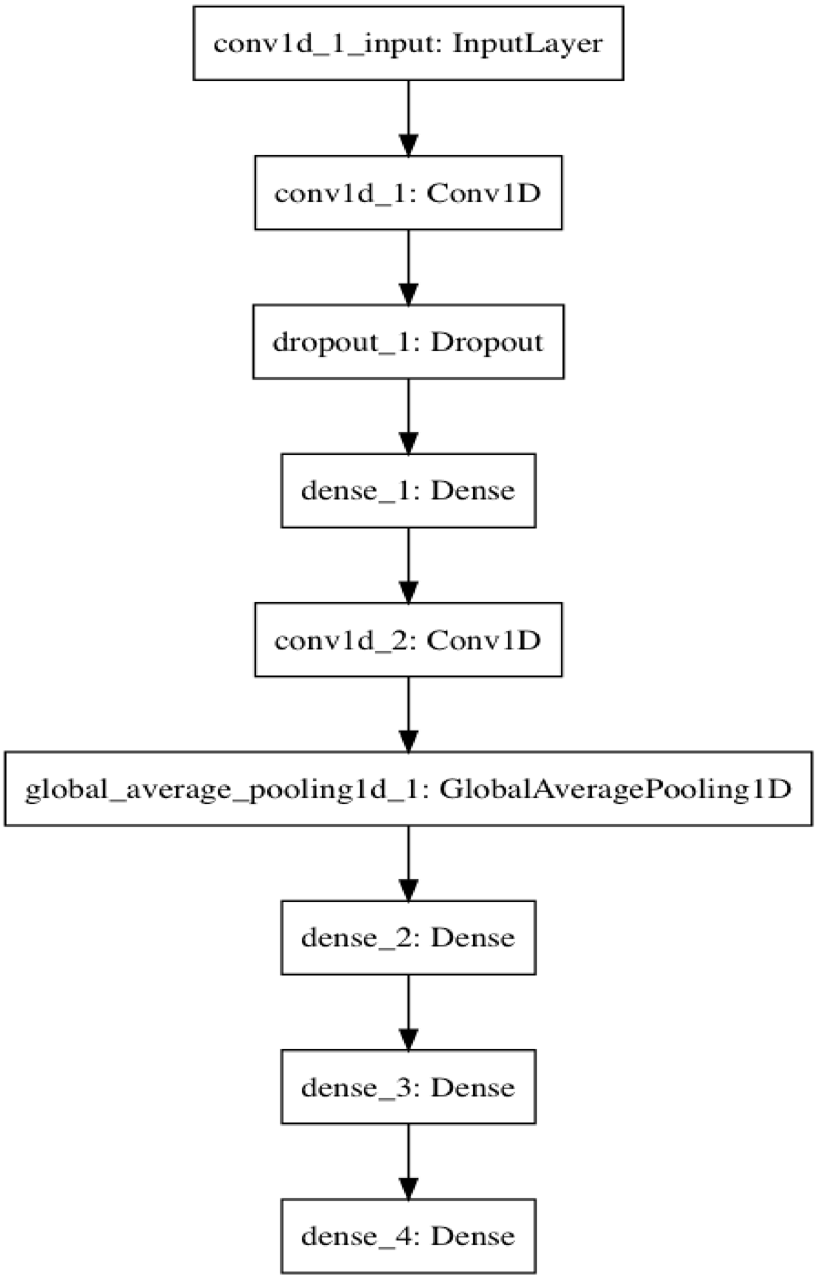
Different combinations of neurons and layers were tested, however, the chosen structure proved to give the best validation accuracy.

3.2.3 Training the model

For training the model, a system running `Ubuntu 16.04.3 LTS`, with an Intel Core i7-6700K, see `https://ark.intel.com/products/88195/Intel-Core-i7-6700K-Processor-8M-Cache-up-to-4_20-GHz`, and 4 Nvidia GeForce GTX TITAN X, see `https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-x/specifications`, was used. The model was split into a parallel model, using keras' `multi_gpu_model()` function from the `keras.util` package. The model was then able to use all 4 GPU's for training, resulting in greater performance.

The model was trained using the ADAM optimizer and the loss is computed using the categorical_crossentropy function. The batch-size was set to 32 and the validation split was set to 20% of the training set.
Early-stopping was also implemented, to stop the training, if the validation-loss was to dip below the lowest value of the 3 preceding epochs. A few different optimizers were tested, such as Stochastic Gradient Descent (sgd) and RMSprop. However, ADAM appeared to give the best results.

Figure 3.6: Visualization using `keras.util.plot_model()` of the chosen model



The training set used for training, is 150, 000 random passwords from the database leak as described in Section 2.1.2 on page 5. The training set was limited in size, due to RAM restrictions because of the large requirements for the chosen model and data representation.
Among the 150, 000 passwords, duplicate passwords were included, to make the model predict these types of passwords more often, as they occur more often among human-chosen passwords. Training the model without duplicates, made the model predict obvious passwords less often than with, which led to the choice of including duplicates.

Figure 3.7: The training and validation accuracy as a function of the number of epochs completed

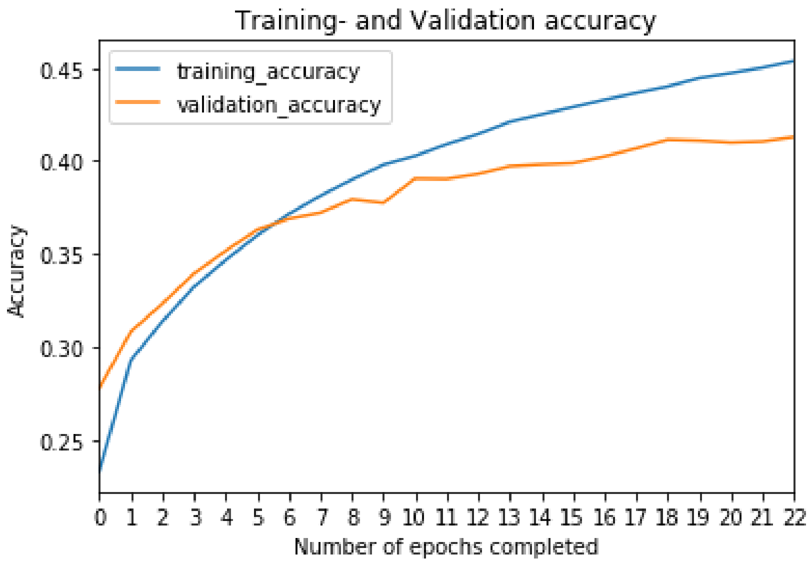
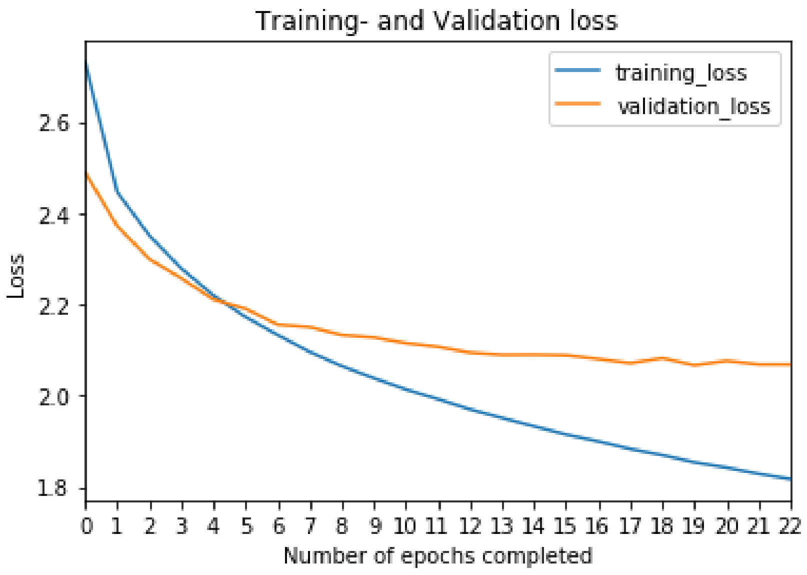


Figure 3.7 is a plot of how both the model's training accuracy and validation accuracy improves as a function of the number of epochs completed, where the accuracy is how well the model predicts the next character given the previous sequence of characters.
The graph shows that the training accuracy starts off slightly lower than the validation accuracy after the first epoch. The accuracy improves quite well during the first 10 epochs, but during subsequent epochs the validation accuracy improves less and less, and by the 22nd epoch, the training process is terminated due to early-stopping. The final validation accuracy is 41.3% and the final training accuracy is 45.4%.

Looking at Figure 3.8, it appears that the loss correlates very well with the accuracy. The loss is reduced a lot quicker in the first 5 epochs, but during the subsequent epochs, the loss converges and actually slightly increases, which triggered the early-stopping.

Figure 3.8: The training and validation loss as a function of the number of epochs completed



3.2.4 Generating Passwords

To generate passwords using the neural network, a method of sampling random intervals between [0.0 : 1.0] uniformly, was chosen.

In practice, this means that the model starts off with an empty string, and then queries the model to list probabilities for each character given the empty string. In order to get random words, yet probable, the sample between [0.0 : 1.0] is used. It iterates through each possible character, and sums up the probabilities from the first character to the k'th character and then to the $k+1$'th character, and checks if the sample value is between those to sums of probabilities, if it is, the character is added to the word and the loops starts over. The following iteration then queries the model for a new list of probabilities given the character chosen in the previous iteration. At some point, the model will predict an 'end-of-password' index, which will terminate the prediction, finalizing the password.

With time, the model should generate many probable, but different passwords, which should look like human-chosen passwords.

Listing 3.1: generate_passwords.py

```python
for i in range(NUMBER_OF_PASSWORDS_TO_GENERATE):
    word = ''
    multiprob = 1.0
    for j in range(20):
        nextpass = False
        x = word2matrix(word)
        x.shape = (1, *x.shape)
        pred = model.predict(x)
        samp = random.random()
        probs = pred[0]
        accumsum = cumsum(probs)
        for k in range(len(probs)-1):
            if k == 0:
                if 0 <= samp <= accumsum[k]:
                    multiprob *= probs[k]
                    word += character_set[k]
                    break
            else:
                if accumsum[k] <= samp <= accumsum[k+1]:
                    if k+1 == feature_size+1:
                        nextpass = True
                        break
                    multiprob *= probs[k+1]
                    word += character_set[k+1]
                    break
                else:
                    continue
        if nextpass:
            break
    if i % 1000 == 0:
        print(str(i) + " took %f seconds" % (time.time()-start_time
            ))
        start_time= time.time()
    lst.append(word+','+str(multiprob)+'\n')
```

The code snippet is seen in Listing 3.1. It iterates through as many passwords as desired, and for each password it initializes two variables, an empty password `word`, and a multiplicative probability `multiprob`.

Then for each character to be predicted, which in this case is up to a maximum of 20 characters, it converts the string, `word`, to a matrix, suitable for the model. It predicts the probabilities for each possible character, given the `word` followed by a sample of a random floating point number between [0.0 : 1.0] (`samp`). In addition to the sample, an array of `probs`'s cumulative probabilities is computed, with numpy's `cumsum()` function

It then moves on to iterating through every possible character's probability of being the next character in the sequence. It checks if the value of `samp` is between the sum of probabilities from the first character up to the k'th character and the sum of probabilities from the first character up to the $k+1$'th character.

If `samp` is between the two values it then checks if k is equal to the end-of-line value, if so, the sequence breaks and the `nextpass` flag is set. The end-of-line value is to indicate that it is more probable to end the password sequence, than to add more characters. Otherwise, it multiplies the character's probabilities with the multiplicative probability (`multiprob`) along with adding the $k + 1$'th character to the sequence (`word`).

When the sequence prediction for each password is over, the password and its multiplicative probability is added to a list, which is then later written to a csv file, of which the probability can be used to sort the list by how probable the password is, potentially resulting in faster guessing.

Generating passwords turns out to be a relatively computationally heavy task. The model can generate about 100 passwords every second, which of course depends on the length of the passwords generated. The function that is taking the longest to compute, is the model's predict function. One optimization that was done to decrease the computation time, was to use numpy's `cumsum()` function to perform the cumulative sums beforehand, instead of summing the array twice for every iteration.

The slow generation speed means that, in order to generate as many passwords that are in the `rockyou.txt` password list, which is just over $14,000,000$, is:

$$14,000,000/100 = 140,000\text{sec} \approx 39\text{hours} \tag{3.3}$$

Bearing in mind that, this is including a quite significant number of duplicate passwords.

A sample of $4,000,000$ passwords was generated using the model. After removing duplicate entries, the actual number of unique passwords, resulted to: $963,691$ passwords. This suggests that model might not have trained on a large enough data-set, or that the model is insufficient for generating that many unique passwords.

3.3 Measuring password strength with neural networks.

With help from the neural network described in Section 3.2 on page 14, it is possible to determine a given password's probability of being predicted by the model, which can be translated into a score or password strength. This is very similar to how Fast, Lean and Accurate [MUS+16] does it.

As a Proof of Concept, a list of 1905 passwords, each with the attributes: length-8 and loweralphanum (recall Section 1.4 on page 3), from German-speaking users of the `Ownedcore.com`-leak, were run through the neural network, and a multiplicative probability for each password was computed.

In Table 3.1 and 3.2 on the next page, are the 10 most probable passwords and the 10 least probable passwords, respectively.
In practice, it means that the model is much more likely to predict the password `oblivion` than the password `xy2zx9nm`, which from a qualitative perspective, seems reasonable.

Because of this, it is possible to measure a password's likelihood of being predicted (its strength), using deep neural networks, where a lower probability is equal to a stronger password.

An interesting thing to note, that was discovered during adjustment of the model, is that the more accurate the model is, the higher the probabilities are for all passwords, suggesting that the model can indeed, be used to predict passwords.

Password	**Probability**
oblivion	0.002909
asdf1234	0.002900
iloveyou	0.001844
abcd1234	0.001365
hello123	0.001359
internet	0.001254
adamadam	0.001227
password	0.001176
warcraft	0.000705
asshole1	0.000549

Table 3.1: Table of 10 worst 8-char passwords with their respective probability score

Password	Probability
gggggggg	2.175686e-30
azbycxdw	6.811967e-32
ipfb5taw	5.245373e-32
69wsxzaq	2.655915e-33
d82q6s8y	2.992962e-34
169575fb	3.261896e-35
g8m468gb	9.193819e-36
ypp88jtj	2.335381e-38
34wrsfxv	2.564595e-42
xy2zx9nm	2.480718e-48

Table 3.2: Table of 10 best 8-char passwords with their respective probability score

3.4 Password cracking using generated passwords

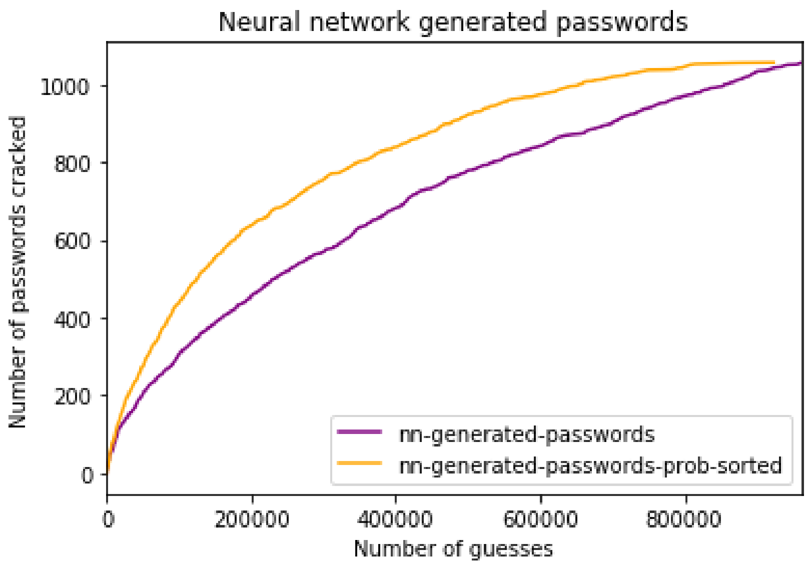
Following the method in Section 3.1 on page 7, it is feasible to use the generated passwords as a dictionary or a password list just like in Section 3.1.1 on page 7. This method will reveal how many passwords of which were generated, is actually used by regular users in the test set from Section 2.1.1 on page 5. Just as in Section 3.1 on page 7 the same list of passwords to crack is used. That way a comparison between standard methods and a machine learning method can be drawn.

In Figure 3.9 on the facing page it is shown how well the generated passwords performed. The line `nn-generated-passwords` represents the complete list of passwords generated from the model without duplicate entries. The password list was able to crack just over $1,000$ passwords out of $100,000$. It may not seem like a lot, but considering that the list consists of just $963,691$ passwords, it is quite reasonable compared to the `rockyou.txt` dictionary from Section 3.1.1 on page 7 of which cracked $15,000$ passwords, but had a list of around $14,000,000$ passwords. It is also worth noting that the model is trained without uppercase characters, which are in the `rockyou.txt` list.

The line `nn-generated-passwords-prob-sorted` represents a password list based on the generated passwords, however, with the difference that came from an idea, to sort the passwords after how likely they are to be predicted, just as how the password strength was measured in Section 3.3 on the preceding page. This method should cause the list to crack the same number of passwords, but crack the more probable passwords faster than an unsorted list. While the model is built to generate probable passwords, the sampling method still has some randomness, such that they are not perfectly sorted by probabilities, but looking at the graph, the line is not completely linear, which is expected from a randomly sorted list.

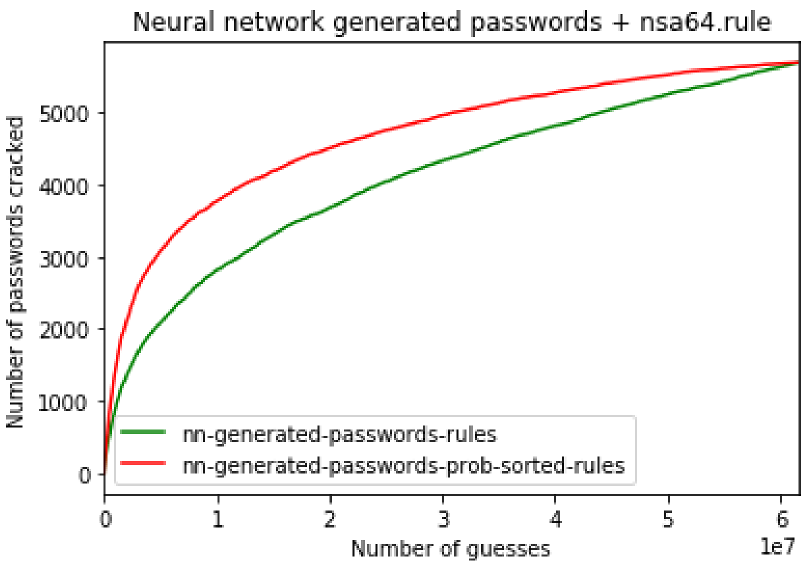
The graph shows that, the list sorted by probability, actually cracks more passwords within fewer guesses, which was what was expected.

Figure 3.9: The number of passwords cracked as function of the number of guesses for the passwords generated from the neural network



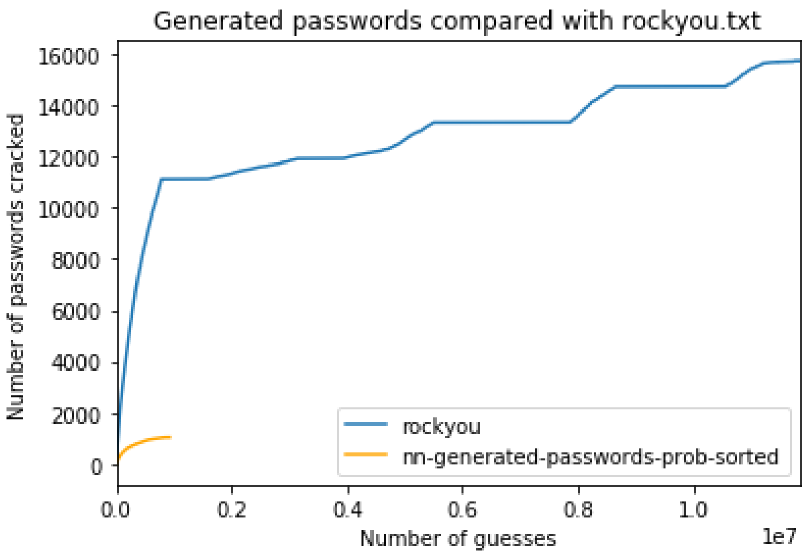
Another method that was tested with the generated passwords, was applying a rule set to the dictionary attack, just like it was done with the `rockyou.txt` list in Section 3.1.2 on page 9. The results are shown in Figure 3.10 on the following page and the method proved to be an expected improvement over the list without a rule set applied. Looking at the line `nn-generated-passwords-rules`, it is seen that, around 5,500 passwords were cracked, but of course with a lot more guesses performed, ending up at around 61,000,000 guesses. The rule set method was also attempted with the generated passwords sorted by the probability, which yielded a similar curve difference as in Figure 3.9.

Figure 3.10: The number of passwords cracked as function of the number of guesses for the passwords generated from the neural network + `nsa64.rule`



In Figure 3.11 on the facing page, a comparison between the generated passwords and the widely known `rockyou.txt` password list, is displayed. The graph shows that, the `rockyou.txt` password list performs a lot better than the generated password in the same number of guesses. The `rockyou.txt` list manages to crack around 11,000 passwords in the same number of guesses that the list of generated password managed to crack just over 1,000, suggesting this model might not be on par with current standard methods of password cracking.

Figure 3.11: The number of passwords cracked as function of the number of guesses for the passwords generated from the neural network compared with `rockyou.txt`



Chapter 4

Discussion and Further work

4.1 Practical use

In practice, the model can have several uses. As shown in Section 3.3 on page 21, it is possible to give a password a score, depending on how likely the model is to predict the given password. This can be used for giving users an indication of how complex or strong their password is before choosing it for any arbitrary website. The time for it to compute the score is quite quick and is suitable for a web server solution.

As demonstrated in Section 3.2.4 on page 18, it is possible to generate, in theory, unlimited passwords, of which are likely to be chosen by humans. However, due to the current implementation, generating passwords is a very lengthy process along with the model generating many duplicate passwords. A different implementation or model might alleviate these issue, to make it more competitive with current dictionaries such as `rockyou.txt`.

Another practical use of which require a bit more tuning to work efficiently, is to use the method of measuring password strength, to sort an already existing password list by probability to increase the rate of which passwords are cracked.

While this use may not prove very useful considering the extremely fast crack speeds for the MD5 hash, it can be very useful for cracking much more computationally heavy hashing algorithms such as bcrypt. Utilizing the hardware used for training the model, the 4 GTX TITANs, the number of bcrypt hashes it can compute per second is $46,000$ compared to MD5 of which it can compute $69,000,000,000$ hashes per second. A huge gap, where the probabilistic sorting can help in cracking passwords using a lot fewer guesses.

Given, that many database leaks also include users' email address or which IP-address they login from, means that it might be desirable to extract passwords related to one type of country. With passwords originating from a specific country, it could be used to train the machine learning model with those country-specific passwords to improve its predictability for that one country.

Furthermore, one could look at just the domain name of a user's e-mail address, then extract only passwords associated with users of that domain name. Take the `dtu.dk` domain for example. It might be achievable to increase the predictability of the model by training it on passwords of users whose e-mail address is associated with that domain. In some database leaks, such as LinkedIn, which is for professional use, many users tend to sign up with work e-mail addresses.

4.2 Improvements to the implementation

4.2.1 Manipulate the random number generator

A suggestion could be to order the probabilities of characters, keeping its mapping the each character and then create a random number generator, that samples a skewed distribution instead of a uniform distribution, to make it generate more numbers closer to 0.

This could allow the model to generate more passwords of higher probability.

4.2.2 Different Machine Learning models or data representations

Further improvements to the model are likely to be available, by using different models, implementations or hyper parameters.

One paper [HGAPC17], suggests to use Generative Adversarial Networks (GAN) for generating passwords, which appears to be quite effective, showing an improvement in a couple of today's standard methods of passwords cracking.

Another paper [XGQ+17], suggests to use Long Short-Term Memory Recurrent Neural Networks (LSTM RNN). The results of this method appear to also perform better than some of today's methods of password guessing.

Different methods of representing the data or sequences, could possibly improve how well the model predicts passwords.

Chapter 5

Conclusion

It is possible with the help of machine learning models, such as artificial neural networks, to predict the sequences of passwords. Because of this, it is also achievable to give a password a strength, based on how likely the model is to predict the given password.

The model can also be used to generate passwords of which humans are likely to choose. The generated passwords can then be used for performing password cracking, using tools such as hashcat with a dictionary-style attack. The generated passwords from the model does not compete very well with current password lists, which have been developed on over many years.

The property of measuring password strength, can be used to order already-known password lists by how likely they are to be predicted by the model. This helps with cracking passwords hashed with a computationally heavy hashing algorithm, such as bcrypt, but also when attempting to bruteforce a login, on a website, with limited login attempts allowed.

The model can be trained relatively quickly, but is very slow at performing the prediction, for measuring password strength, generating passwords and sorting password lists by probability. This limits its practical use for large password lists.

Bibliography

[AAB+15] 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. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[C+15] François Chollet et al. Keras. https://github.com/keras-team/keras, 2015.

[Gos17] Jeremi Gosney. Nvidia gtx 1080 ti hashcat benchmarks, March 2017. https://gist.github.com/epixoip/973da7352f4cc005746c627527e4d073/.

[HGAPC17] Briland Hitaj, Paolo Gasti, Giuseppe Ateniese, and Fernando Perez-Cruz. Passgan: A deep learning approach for password guessing, 2017.

[MUS+16] William Melicher, Blase Ur, Sean M. Segreti, Saranga Komanduri, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Fast, lean, and accurate: Modeling password guessability using neural networks. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 175–191, Austin, TX, 2016. USENIX Association. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/melicher.

[NSA16] NSAKEY. Password cracking rules and masks for hashcat that i generated from cracked passwords., June 2016. https://github.com/NSAKEY/nsa-rules.

[wea] weakpass.com. The most complete compilation of wordlist's - more than 1500 in one. contains near 8 billion of passwords with length from 4 to 25. https://weakpass.com/wordlist/1861.

[Woo14] Robin Wood. Pipal, the password analyser, 2014. https://github.com/digininja/pipal.

[XGQ$^+$17] Lingzhi Xu, Can Ge, Weidongg Qiu, Zheng Huang, Zheng Gong, Jie Guo, and Huijuan Lian. Password guessing based on lstm recurrent neural networks, July 2017.