CS782 Final Project implementing PassGAN

Carson Forsyth
George Mason University
cforsyt@gmu.edu

Michael McNamara
George Mason University
mmcnama@gmu.edu

1. Abstract

Our project was to implement the paper PassGAN: A Deep Learning Approach for Password Guessing by Hitaj, 2019 et al. [2]. This paper discusses the use of GANs and their potential to outperform rule-based password-cracking tools such as HashCat and John the Ripper. We implemented their approach by utilizing the IWGAN architecture [1] and then modifying it to match the architecture and needs of PassGAN. The goal of this is to reproduce the results described in the paper to demonstrate our understanding of all its aspects.

2. Introduction

Passwords are a very common security measure utilized by people and organizations. They come in many forms and degrees of security, but the most common are character-based passwords that usually allow the user to access a private account or protected system. Everyone has at least one password that they have to remember, but the vast majority of people have multiple to dozens of passwords to remember. Ideally, each account would have its own unique password that is long and complex. Long in terms of the number of characters and complex in terms of the variability of different characters used and how random it looks. One of the issues with passwords is that they require the user to memorize them all and to facilitate this, most people choose easy-to-remember passwords which make them less secure. This usually involves combining a word or two together with some numbers and maybe a special character to create a password. Doing all these things enable password attacks to quickly compromise a password. Character-based passwords are not the most secure, but due to them requiring no extra hardware, it has become the standard for security in most areas.

Passwords are susceptible to a variety of different attacks including brute-force, dictionary attacks, and rainbow tables. PassGAN is being presented as a way of helping researchers understand what are the limits to password guessing. If the limits of password guessing are better understood, then we would have the ability to understand what constitutes a good password or not.

3. Related Work

3.1. Other password cracking with GANs

Just last year there was a new paper published which improves upon PassGAN's results. The paper is titled Password Guessing Based on GAN with Gumbel-Softmax by Zhou, 2022 et al. [5] This paper aims to outperform PassGAN by first modifying the network structure by replacing the generator with a long short-term memory (LSTM) based network which uses multiple convolutional layers. Second, they implement Gumbel-Softmax with temperature control for training [5].

The results of this paper are exciting and very interesting. Not only because it outperforms PassGAN in their evaluations (which are similar to PassGAN by seeing if a password that is generated appears in the testing data) but also because they introduce temperature hyperparameters which allow one to alter whether more diverse passwords are generated or higher quality passwords are generated. Higher-quality passwords are measured by taking the number of unique passwords and dividing them by the number of matched passwords. The higher the N_{match} the lower the quality of the generated sample.[5]

$N_{match} = \frac{numberofuniquepasswords}{numbermatchedpasswords}$

4. Approach

We used the WGAN-GP implementation by Ishaan Gulrajani as a starting point for our implementation.[1] The architecture has been modified to meet the architecture that was discussed in the PassGAN paper. For reference, it can be viewed in figures 1 and 2. As seen in the figures provided, the generator starts with a linear layer followed by 5 residual blocks and ends with a 1D convolution and a transpose operation into a softmax activation function. The discriminator is similar to the generator in terms of architecture except that it is in the reverse order. It starts with a 1D convolution, then 5 residual blocks, and then reshaped and passed through a linear layer. We then ran two main experiments in an attempt to recreate the results observed in the

paper. The two experiments include confirming the number of matched passwords per checkpoint and confirming the number of matched passwords per number of passwords generated after fully training the model. The results of these experiments will be discussed in the results section below.

4.1. data

The two password data sets used in the PassGAN paper were the RockYou password dataset and the LinkedIn password data set. We encountered difficulties acquiring these data sets in the same manner as the original authors. We were able to acquire the RockYou data which originally contained 14 million unique passwords for 32 million accounts, but we were only able to find a list of the 14 million unique passwords and not their counts [3]. For the LinkedIn data set, we searched but were unable to find it available. The link in the references section of the PassGAN paper was not working and we were unable to find it elsewhere.

The RockYou data set contains about 14 million unique passwords for 32 million accounts. Due to the company's poor security practices, they all were stored in plaintext and thus very susceptible to being compromised. Of the 14 million passwords, any passwords of length greater than 10 were removed, leaving approximately 10 million which fit this criteria. From there, 80% of them were used for training and the other 20% were used for testing. This yielded about 8 million passwords for training and about 2 million for testing.

4.2. IWGAN

The core of this model is the WGAN-GP, referred to as the IWGAN in the PassGAN paper [2]. This is a variant of the original Generative Adversarial Network (GAN) originally designed by Ian Goodfellow that is presented as an improved version due to its new loss function and gradient penalty. We used the IWGAN model described in the IWGAN paper as the basis for our implementation and then expanded on it and modified components to ensure that it matched the model described in the PassGAN paper and thus would adequately suit our needs.

To implement this we opted to use the implementation of the paper Improved Training of Wasserstein GANs by Ishaan Gulrajani as our starting point. [1] To do so we had to modify the structure of the project to operate properly in a notebook environment, we also had to modify how data was read in, passwords were generated (correct amount and at correct intervals which caused many issues which we will discuss in the Challenges faced section) and the architecture of the model. Furthermore, the code was originally written in TensorFlow 1 so to be able to use it we had to update the functions used. We also went through and ensured that all the architectural choices and hyperparameters are consistent with the architecture described in the PassGAN paper.

4.3. Generator

The generator as shown is comprised of a linear layer that contains 128 units as specified by the PassGAN paper, followed by 5 residual blocks and ends with a 1D convolution which has 128 filters, and finally a transpose operation into a softmax activation function. These parameters such as the number of units or filters can be adjusted to increase or decrease training time as well as adding or removing residual blocks. However due to extremely long training times to simply reproduce results we did not have time to experiment with additional structures.

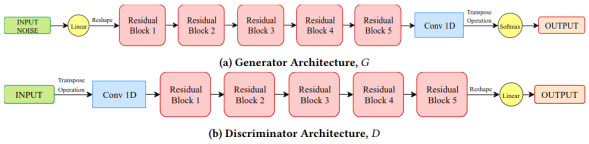

Figure 1. PassGAN generator and discriminator architecture

4.4. Discriminator

The Discriminator follows a very similar structure as the generator however the transpose operation along with the 1D convolution comes before the residual blocks. Once again the 1D convolution has 128 filters. This 1D convolution is then followed by 5 residual blocks and ends with a reshape operation into a Linear layer. The linear layer has 128 units once again. Similarly to the generator, the parameters such as the number of 1D convolutions along with the number of filters and units in the various layers can also be adjusted here.

4.5. Residual Block

The residual block that is detailed in figure 2 is fairly simple. It consists of a ReLU activation function followed by a 1D convolution which is then followed by a ReLU activation function and another 1D convolution. The output is then multiplied by 0.3. The authors do not explain why they multiply by 0.3 however it is included in our implementation. There is also a shortcut connection from the input to directly after the 0.3 multiplication.

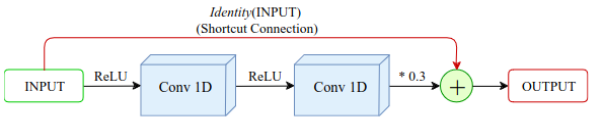

Figure 2. Residual Block

4.6. Jensen-Shannon divergence

As discussed during our paper presentation the act of generating a large number of passwords and seeing how many appear in the test set does not seem like a very intuitive way of measuring how effective a model is at generating passwords. This is why Jensen-Shannon divergence is used in this implementation of PassGan, it is similar to Frechet Inception distance which was used for assignment 2, however, we are not comparing generated images but rather distributions of passwords. Therefore we must use some metric which can handle two probability distributions.

The Jensen-Shannon Divergence, also known as JSD, is a symmetrical implementation of the Kullback-Leibler Divergence. Meaning that it will produce the same results regardless of which distribution is considered the reference distribution [4]. The equation used to calculate JSD is shown here.

$JSD(P||Q) = \frac{1}{2}KL(P||M) + \frac{1}{2}KL(Q||M)$
$where M = \frac{1}{2}(P + Q)$

M is the "average" of the distribution and KL is the Kullback-Leibler divergence between P and Q which can be defined as.[4]

$KL(P||Q) = \sum p(i) * \log(\frac{p(i)}{q(i)})$

Where the sum is taken over all values of i, and $p(i)$ and $q(i)$ are the respective probabilities of i in the P and Q distributions.[4]

In JS divergence the lower the value the more similar the two probability distributions are to one another where a value of 0 means that the distributions are identical. The results of JS divergence are bounded between 0 and 1.[4]

This was implemented by initially taking from the training data and then computing the JS divergence against another randomly sampled data set of the same size from the training data. This was done 4 times with n grams of size 1, 2, 3, and 4 to establish a baseline. Every 1,000 iterations the samples that were generated were evaluated on the JS divergence on 4 of the baseline sampled data sets that were initially generated. We will discuss the findings of tracking the JS divergence over training in the results section.

4.7. Finding matches

Finding the matches was a fairly straightforward task after generating all the necessary passwords for the experiments that we recreated. We first had to load in the RockYou test set and read all passwords into a list. We then converted this list to a set to remove any duplicates. We did the same steps with all of our generated passwords and computed the intersection of both sets (the RockYou data set and generated password set we were testing). This saved both compute and allowed us to remove any duplicates from either set that could inflate our matching numbers.

4.8. Challenges faced

There were many challenges faced when implementing this paper. First was the sheer amount of data that needed to be handled both when reading in RockYou and also generating passwords. We had to be clever about how we managed both our RAM and also VRAM to be able to correctly generate 10^8 passwords without crashing our kernels or even worse our own machines. When generating 10^8 passwords we were consuming over 30 GB of RAM alone simply holding the generated passwords in a list before even writing them to a text file. When finally written to a file the files alone were over 1 GB large only containing passwords that were of length 10 or less. This resulted in us having to manually release memory after the passwords were generated since neither of us had more than 32 GB of memory available. However even after manually releasing the memory our kernels would still crash, this led us to incorporate a slightly slower procedure of writing smaller chunks to the files over time to avoid using too much ram.

On top of dealing with many memory issues, we also had extremely long training times even with a GPU. Training the model to what the authors did took just over 12 hours, we were very fortunate to achieve this after only having to restart our training because of dead kernels a few times.

5. Results

Our goal was to recreate two main portions of this paper first we wanted to see if we could reproduce the results that are observed in Figure 3. This table shows the number of passwords matched for specific checkpoint numbers. We were very intrigued by the dip that is observed around checkpoint 145,000 and wanted to see if we would experience the same behavior. Second, we wanted to reproduce the results that are observed in Figure 5. This table shows the number of passwords that are matched in the RockYou data set based on how many passwords are generated. As mentioned in our data section we have a modified version of the RockYou data set and therefore were unable to get an exact recreation of the paper.

We were able to recreate a very similar pattern of matched passwords per checkpoint. Our results are in figure 4 As you can see in figure 4 we have a large jump in performance from checkpoint 5,000 to checkpoint 15,000. Following this jump there is a steady increase in matches until checkpoint 145,0000. Our model did not experience the same dip in performance at checkpoint 145,000 as is seen in 3 but we attribute this to the difference in data sets.

For reference, we have included a table that shows the percent matched per checkpoint. These numbers seem low but when considering the difference in data set size they are comparable to those seen in the original paper.

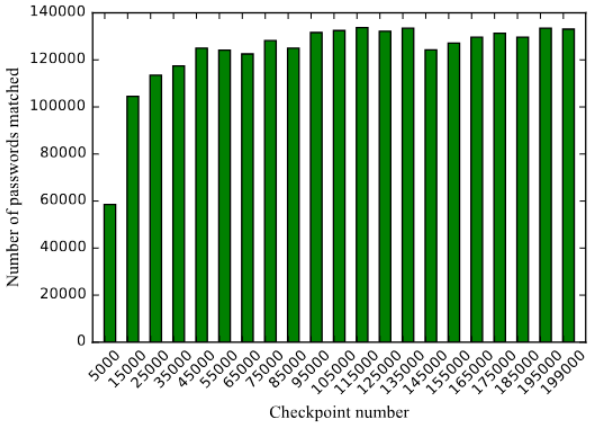

Figure 3. Number of unique passwords generated by Pass-GAN on various checkpoints, matching the RockYou testing set. The x-axis represents the number of iterations (checkpoints) of PassGAN's training process. For each checkpoint, we sampled 10^8 passwords from PassGAN.

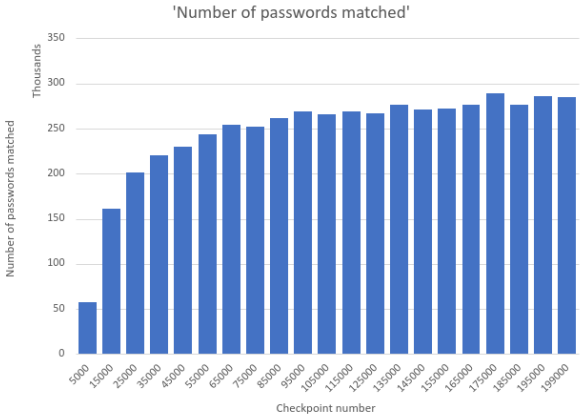

Figure 4. Our recreated number of passwords generated by Pass-GAN that match passwords in the RockYou testing set. Results are shown in terms of unique matches.

Checkpoint	Percent Matched
5000	0.029232473
15000	0.081378579
25000	0.101671662
35000	0.111594264
45000	0.11610996
55000	0.122913077
65000	0.128327565
75000	0.127385805
85000	0.132342809
95000	0.13595668
105000	0.134124722
115000	0.135914723
125000	0.134690384
135000	0.139836954
145000	0.137063235
155000	0.137713823
165000	0.139915307
175000	0.145997065
185000	0.13957409
195000	0.144366804
199000	0.14403418

Percent of passwords matched per checkpoint

Passwords Generated	**Unique Passwords**	**Passwords matched in testing set, and not in training set (1,978,367 unique samples)**
10^4	9,738	103 (0.005%)
10^5	94,400	957 (0.048%)
10^6	855,972	7,543 (0.381%)
10^7	7,064,483	40,320 (2.038%)
10^8	52,815,412	133,061 (6.726%)
10^9	356,216,832	298,608 (15.094%)
10^{10}	2,152,819,961	515,079 (26.036%)
$2 \cdot 10^{10}$	3,617,982,306	584,466 (29.543%)
$3 \cdot 10^{10}$	4,877,585,915	625,245 (31.604%)
$4 \cdot 10^{10}$	6,015,716,395	653,978 (33.056%)
$5 \cdot 10^{10}$	7,069,285,569	676,439 (34.192%)

Figure 5. Number of passwords generated by PassGAN that match passwords in the RockYou testing set. Results are shown in terms of unique matches.

Passwords Generated	Unique Passwords	Passwords Matched (1,978,211 samples)
10^4	9,999	121 (0.006%)
10^5	99,879	1,269 (0.064%)
10^6	989,481	11,357 (0.574%)
10^7	9,373,669	75,760 (3.830%)
10^8	79,799,525	262,081 (13.248%)
10^9	590,760,905	559,736 (28.295%)

Table 1. Number of unique passwords generated when sampling different magnitudes of passwords

As mentioned, we also wanted to reproduce the results in Figure 5 from the original PassGAN paper. Our results are shown in table 1. As shown, we achieved somewhat close results to the originals. Generally, we had more unique matches compared to the original paper in both of our experiments. We were unable to retest all the experiments for the larger-sized sampling sets. We were limited by both compute resources and time to generate and process the well over a hundred gigabytes of text data needed to perform all the tests the original authors did. We believe that the larger amount of unique passwords stems from the training set having only unique passwords and so the model learned to create new ones accurately instead of repeating the same ones multiple times like how the original RockYou data set had 18 million duplicate passwords.

Along with testing the uniqueness of different magnitudes of samples, Table 1 also shows that we tested how many matched results in the test set the same as the authors. Our results show a similar trend to that of the authors of the number of matches increasing over time. Through these different tests, we have shown that our implementation of PassGAN is very similar to that of PassGAN and thus is a successful reproduction of the paper.

5.1. JS Divergence

The JS divergence graphs that were created while training the model can be found in the appendix in figures 6, 7, 8, and 9. We believe this is a better metric to measure how well the model is generating passwords since it represents a quantitative metric that can be used to measure the differences between two probability distributions. As seen in all of the figures mentioned before the model seems to reach a steady level of around 60,000 iterations in figures 7, 8 and 9 while in 6 it takes slightly longer to stabilize around iteration 125,0000. Depending on how closely one wants their generated samples to resemble the training data using JS divergence during training would allow one to stop when they believe their generated samples have reached a satisfactory point and be able to quantify it with a JS divergence value.

Each of the four graphs in the appendix shows the JS divergence calculated using different n-gram models. The four n-gram models (1-4) 6, 7, 8, 9 show the divergence between the generated output and the training data as the divergence between two distributions. For 1-gram, 2-gram, and 3-gram, our implementation had little divergence from the training set. Our 4-gram divergence was quite high, which could have to do with only generating length 10 passwords, and so there is so much more variability among them. The authors did not report their own JS divergence values and so we were unable to compare, but we believe that on their own, our values are meaningful and good. Having these low divergences across multiple n-gram models shows that PassGAN is not an n-gram-based model and thus can learn patterns that form in n-grams but also ones that do not.

6. Conclusion

In conclusion, we were able to recreate the results of this paper as seen in figures 4 with a smaller data set we observed very similar behavior to the original paper. Furthermore, we saw similar results in both the number of matches as well as the rate of new passwords. Despite having a somewhat different data set compared to the original, we were able to proudly accomplish our goal of reproducing the results that we set out with. We enjoyed this project since it gave us a deeper understanding of GANs and their variants as a whole. Additionally, we gained valuable experience with really studying a model and researching and learning about all its ins and outs. Furthermore, it was a very interesting topic and we would like to work more on it. We find the new related work very interesting since it outperforms PassGAN by replacing the generator with an LSTM.

References

[1] I. Gulrajani. Improved wgan training, 2017.

[2] B. Hitaj, P. Gasti, G. Ateniese, and F. Perez-Cruz. Passgan: A deep learning approach for password guessing, 2019.

[3] R. Mutalik, D. Chheda, Z. Shaikh, and D. Toradmalle. Rockyou, 2021.

[4] udit. The jensen-shannon divergence: A measure of distance between probability distributions, 2023.

[5] T. Zhou, H.-T. Wu, H. Lu, P. Xu, and Y.-M. Cheung. Password guessing based on gan with gumbel-softmax. *Security and Communication Networks*, 2022:5670629, Apr 2022.

7. Appendix

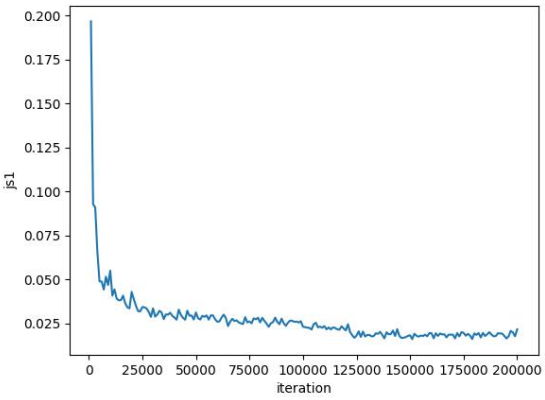

Figure 6. JS divergence over training iterations with 1-gram.

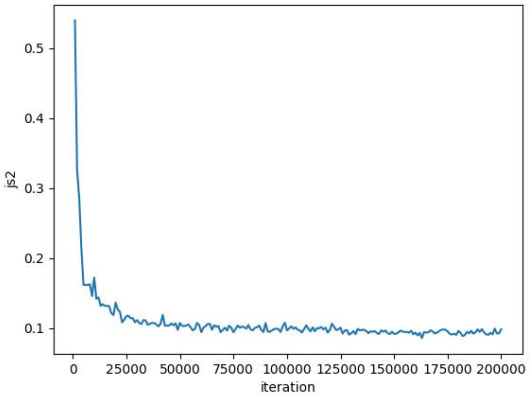

Figure 7. JS divergence over training iterations with 2-gram.

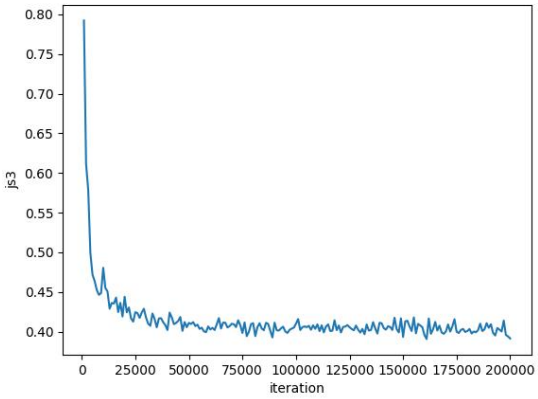

Figure 8. JS divergence over training iterations with 3-gram.

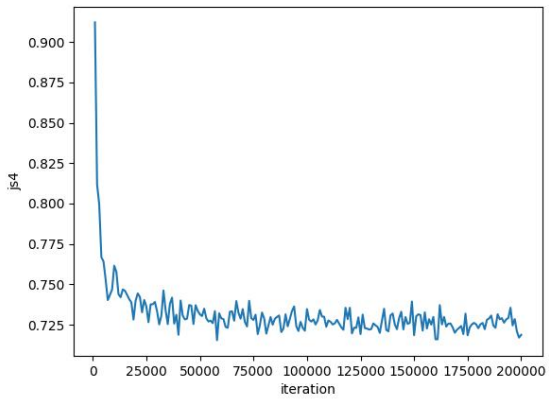

Figure 9. JS divergence over training iterations with 4-gram.