# Variational Autoencoders for Password Prediction and Scoring

Noah Thurston
Dept. of Electrical and Computer Engineering
The University of Arizona
Tucson, AZ 85721-0104
noahthurston@email.arizona.edu

***Abstract*—The average citizen must create and remember dozens of online credentials just to survive in modern society. At the same time, they are pestered to follow good security practices by using unique passwords on every site. This pressure to create credentials that are both unique and memorable forces them to consistently commit the same lame-duck compromise of only creating a single truly unique password, but using it to generate a diaspora of marginally different offshoots to use all around the web. We demonstrate the weakness of these small password manipulations by leveraging variational autoencoders to generate new passwords similar to a given seed. Then we use the same model as an engine for scoring the strength of passwords with reconstruction error. By understanding and modeling the patterns humans follow during password generation we hope to strengthen authentication on the web.**

***Index Terms*—Variational Autoencoders, Passwords, Password Cracking, Security**

## I. Introduction

Passwords are the cornerstone of authentication on the web. Even with the rising popularity of hardware based authentication methods such as Yubikeys, passwords still reign as the go-to method for authentication. Unfortunately, users still regularly reuse passwords or only minimally change them from site to site. This creates a large security threat we aim to highlight in this paper. Overall, our goal is to demonstrate how the latent space of variational autoencoders can be leveraged to predict passwords and measure their strength.

First, we focus on the black-hat goal of being able to generate passwords similar to a given seed password. We define passwords as being "similar" not in the strict Manhattan distance sense, but rather by the idea that they only differ in the same predictable ways that users change their passwords. Predictable changes that users often make, as noted by Das et. al. [1], include adding a number to the beginning or end, changing capitalization, appending punctuationalization, or simply making the new password a substring of the original. In fact it was found that 19% of passwords are substrings of one another. And when users were asked to create a password on the spot, 43.4% of the passwords created consisted solely of a name or word with numbers or symbols appended or prepended [2]. The results point to the reality that many passwords may be unique in a strictly mathematical vector interpretation, but are predictably similar to other passwords used. Our first goal was to be able to replicate these predictable patterns using variational autoencoders.

We were able to achieve this by training a variational autoencoder to map a set of known plaintext passwords to a latent space. Within this latent space, small perturbations can be added such that the resulting vectors when decoded yield passwords with changes similar to those made by humans. These latent representations can also be leveraged to interpolate betweent two passwords that are known to be belonging to the same individual such that other similar passwords can be generated.

Second, we create a white-hat tool to aid in measuring the strength of passwords. Many websites have password strength requirements, but these are often simplistic in nature. One example is the "Strong1" requirement used by Chase Bank that specifies that passwords must "Contain 7-32 characters with at least one letter and one number" [1]. Our ideal engine would go beyond these simple checks to ensure that passwords are not predictably similar to passwords likely used by other users.

To create an engine like this, variational autoencoder reconstruction error is observed. We show that user passwords from the compromised data set are easily reconstructed with low error, and therefore are scored as poor passwords. Meanwhile, passwords with characters randomly drawn from a character distribution estimated off of the compromised data set make for much stronger passwords with higher scores. And lastly, passwords that score the highest are comprised of characters drawn uniformly randomly from the complete alphabet of possible characters. Although these results do not provide strong guarantees of the models ability to generate and identify strong and weak passwords, they do demonstrate how variational autoencoders can be a powerful tool in recreating how human psychology tweaks credentials in an attempt to balance security and convenience.

## II. Background

Our goal is to generate passwords that are new but realistically similar to those that already exist in our data set. This motivates the need for a generative model. Generative models can estimate the distribution of our data set $X$ so that we can sample from it. Variational autoencoders are generative models that work by creating functions that map samples from a data

set to and from a latent space that fits a goal distribution [3] [4].

To formulate this mathematically we first start by stating we have a high-dimensional latent space of representations called $Z$ from which we can sample from. Then we wish to create a function with parameters $\theta$ such that $f : Z \times \theta \mapsto X$. This allows for any sample from our latent space to be mapped into the space of our original data set $X$. But not only should any sample $x$ generated from $z$ exist, it should also look like a real password belonging to our original set $X$. To do this, we maximize the probability $P(X)$, so that with our generative model the probability of us generating samples from our original data set is high.

We can calculate $P(X)$ using the law of total probability by integrating over the entire latent space as:

$$P(X) = \int P(X|z;\theta)P(z)dz. \tag{1}$$

In this equation $P(X|z;\theta)$ is the distribution of outputs from our function as $Z \mapsto X$ using the parameters $\theta$. We want this to be a multivariate Gaussian distribution (not just a unit impulse), so that when mapping from the latent space we can get multiple unique samples that all look like samples from the original data set, instead of just a single deterministic choice from the original data set.

Now to maximize $P(X)$ we could estimate the integral as a finite sum over the space of $z$ values as:

$$P(X) \approx \frac{1}{n}\sum_i P(X|z_i) \tag{2}$$

But this is still intractable. So instead we create a function $Q(z|X)$ to give us a distribution of likely z values for any $X$. This can be thought of as an encoding layer that maps our data space to our latent space. Next we use the definition of KL-divergence to find the divergence between the distribution of $z$ values created using our $Q(z|X)$ function and the distribution $P(z|X)$:

$$D[Q(z|X)||P(z|X)] = \mathbb{E}_{z\sim Q}[logQ(z|X) - logP(z|X)] \tag{3}$$

Using Bayes theorem to divide up $P(z|X)$, and rearranging, we come to:

$$\begin{aligned} logP(X) - D[Q(z|X)||P(z|X)] = \\ \mathbb{E}_{z\sim Q}[logP(X|z)] - D[Q(z|X)||P(z)] \end{aligned} \tag{4}$$

This equation is key in that it shows that by maximizing the right side we are are maximizing the log probability of $X$ while minimizing the KL-divergence between the distribution created by our encoding function and the true distribution of $P(z|X)$.

Maximizing the second term is simple in that it is only the divergence between the random variable $z$ (which we defined as $z \sim N(0,1)$) and the output from our encoding function $Q(z|X)$. Meanwhile, maximizing the first term is done by letting a single stochastic sample from $z$ be an estimate of $E_{z\sim Q}[logP(X|z)]$. This term then becomes the log probability of our decoding function output.

Up to this point we have discussed the functions $Q(z|X)$ and $P(X|z)$ in general, but practically they are implemented and tuned as neural networks. This is done by having $Q$ be a dense network for 'encoding' where samples are the input and the output is the mean and variance parameters to a normal distribution. These defined parameters are then used to calculate the KL-divergence between $Q(z|X)$ and the normally distributed $P(z)$. Then a single sample is drawn from a fixed distribution $N(0,1)$ and reparameterized using the mean and variance output from the 'encoding' function $Q$ to produce a single latent vector $z$. We sample from $N(0,1)$ and reparameterize it instead of sampling directly from $N(\mu_Q, \sigma_Q)$ so as to avoid taking the gradient of a stochastic function.

This latent vector $z$ is the fed through another dense network to produce the distribution $P(X|z)$. After this, the gradient of both the KL-divergence term and the log probability term are calculated for optimizing the $Q(z|X)$ and $P(X|z)$ neural networks.

## III. Problem Formulation

Humans follow many fuzzy but predictable patterns when generating passwords [1]. But describing these transformations in detail and implementing a tool to apply all these different transformations would be tedious. So instead we trained a variational autoencoder on a set of known plaintext passwords.

The first reason this was done was that VAEs work to maximize the posterior probability of the data set provided, but at the same time they are restricted by the size of the latent encoding. To succeed under this constraint, the model must learn an efficient encoding that focuses on understanding only the patterns popular to the given data set. This leads to patterns that are unpopular being poorly reconstructed when they are fed through the autoencoder. Therefore we can use the sign of a high reconstruction error as a signal that the input password was novel, and therefore provides a higher level of security than a password featuring more popular patterns.

The second reason for using VAEs is that they are constrained to map the data distribution to a fixed Gaussian distribution in the latent space. This allows us to sample from the latent distribution freely and with confidence that any popular point we pick from the Gaussian distribution in the latent space maps to a password that is similar to those in our data set. This forms the basis of a generative model we can use to generate novel passwords.

## IV. Implementation

The data used for training and testing was downloaded from the SecLists GitHub repository by user danielmiessler [5]. The set used contains one million popular plain text passwords found on the web. To clean the data, first all short passwords have underscores appended until they are ten characters long. Then, longer passwords are truncated to contain only the first ten characters. From there, a one-hot encoding dictionary is formed, so each character is assigned to a single index in the vector. In the data set, 95 unique characters were found. Of these 95 characters, 90 are encoded in the one-hot

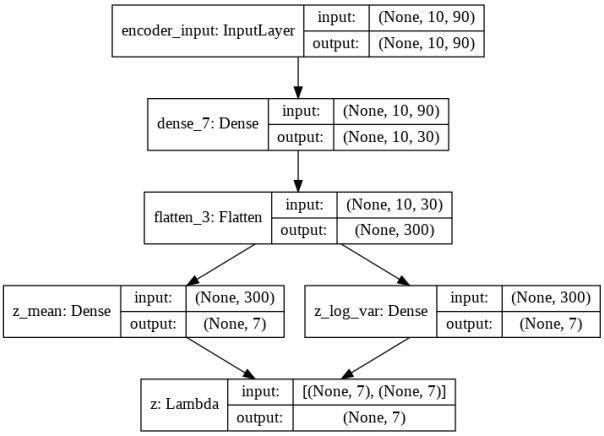


Fig. 1. A flow graph of the encoding network to produce $Q(z|X)$, implemented as a dense feed-forward neural network in Keras.

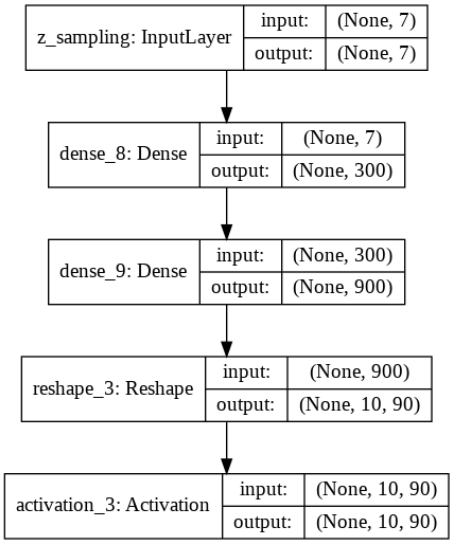


Fig. 2. A flowgraph of the decoding network to produce $P(X|z)$, implemented as a dense feed-forward neural network in Keras.

vector, and the remaining five are replaced by underscores. Finally, each password was converted to a neural network digestible representation as an ordered sequence of ten one-hot vectors, with each vector representing a single letter. This form of representation was chosen for multiple reasons. The first being that unlike the popular "bag of words" method, using a sequence of one-hot vectors allows for the order of letters in words to remain represented. And second, by padding or truncating passwords to be only ten characters, it is possible to feed the data into a neural network without the need for dynamically unrolling recurrent nodes depending on the length of individual passwords.

The variational autoencoder was implemented using Keras and the provided documentation [6]. It uses dense layers to map the input sequence into the latent space, where it can be stochastically sampled, and then subsequently decode it back into the original data space. The encoder takes an ordered input of ten one-hot vectors each 90 bits long, followed by a 300-node dense layer. Then followed by a pair of 7-node dense layers to produce the latent space mean vector and log-variance vector. These layers are then combined by sampling from a unit normal multivariate Gaussian distribution and scaling and shifting the sample by the specified log-variance and mean vector. This results in the sampled 7-dimensional latent encoding. To map this encoding back to the data space the decoder uses the 7-d latent representation as input, followed by a 300-node dense layer, then a 900-node dense layer. The output from this final layer is reshaped to match the original input shape of 10x90 per sample, then placed through a softmax activation. This softmax ensures each 90-long output forms a distribution over the 90 possible character inputs. As described earlier, variational autoencoders are trained using a loss function based on both maximizing the log probability of the training set samples and an estimated KL-divergence between the $z$ distribution and a unit normal distribution. In this implementation, the log probability of the data set is maximized by calculating the categorical cross entropy between each letter's one-hot vector and the predicted distribution for that letter's position. The dataset was divided 90-10 for a train-test split, creating 900,000 samples for the training set and 100,000 for the test set. Adam optimizer was used during training and a total of 10 epochs were performed.

For the complete implementation visit the projects GitHub repository [7].

## V. Results

After training the VAE, we wanted to assess the models ability to generate passwords and measure their strength. But first we wanted to probe the latent space to get an understanding of whether or not it learned well enough to generate realistic samples. To do this we first performed two tests; the first attempts to evaluate how well the model can encode and decode passwords. To do this we sampled from the holdout test data, encoded the passwords into the latent space and then subsequently decoded them. The results of 15 random samples are shown in figure 3.

The results show that the model has an efficient latent representation to encode most information from the password. And if we encode and decode all passwords from the test set we find on average only 1.65 characters are decoded incorrectly. This performance on the test set is convincing that the autoencoder was able to efficiently represent patterns in the passwords using a lower dimensional latent space.

Now we can attempt to generate realistic passwords using our model. We did this two different ways, which yielded different results. The first way was simply sampling vectors that are close to the latent representation of a given seed password. This second was to interpolate between two encoded seed vectors within the latent space.

To do this we first encode a password from our test set and map it to the latent space. Then we create new latent encodings similar to the original by adding a random Gaussian noise

| Sample | Input | Output |
|---|---|---|
| 1 | diamanwj__ | diamando__ |
| 2 | jemelnik__ | jemelnom__ |
| 3 | wb7zhg8___ | wb7zht1___ |
| 4 | frg2608___ | frg2600___ |
| 5 | 24531706__ | 24531006__ |
| 6 | 84lumber__ | 84lumber__ |
| 7 | biggdog___ | biggdog___ |
| 8 | luigi090__ | luigi000__ |
| 9 | choirboy__ | choirkoy__ |
| 10 | erinwest__ | erinwern__ |
| 11 | 28081962__ | 28081982__ |
| 12 | slim1234__ | slim1234__ |
| 13 | fishes77__ | fishes77__ |
| 14 | xn2287y___ | xn22821___ |
| 15 | kingsix___ | kingsit___ |

Fig. 3. The table above shows passwords from the test set as they are input to the VAE as well as the reconstruction output by the VAE.

| | slim1234__ | xn2287y___ |
|---|---|---|
| 1 | blim1237__ | xn22801___ |
| 2 | bliy1234__ | xn22801___ |
| 3 | blim1234__ | 3n22827___ |
| 4 | slim1234__ | xn22801___ |
| 5 | plim1234__ | xw22821___ |
| 6 | plim1634__ | xn22821___ |
| 7 | blim1234__ | 3n22801___ |
| 8 | slim1234__ | xn22801___ |
| 9 | blimo237__ | xw22r21___ |
| 10 | sliy1634__ | xn22r01___ |
| 11 | bliy1334__ | xn2282____ |
| 12 | slim1237__ | xn22801___ |
| 13 | sl4m1237__ | xn22801___ |
| 14 | slim1234__ | 3n22821___ |
| 15 | blim1234__ | xn22801___ |

Fig. 4. The table above shows the decoded passwords after Gaussian noise is added to the latent vectors belonging to ”slim1234__” and ”xn2287y___”

vector to the original latent encoding. This results in new latent vectors close to the original within the latent space. Lastly, we decode these new vectors using the decoder to see what the password looks like in the original data space.

The results shown in 4 from these two limited examples are only somewhat promising. The autoencoder was able to make small perturbations to the password, but some were more realistic than others. For example, changing the 4-digits in ‘slim1234’ from ‘1234’ to ‘1334’ seems realistic, but changing the spelling of the base word from ‘slim’ to ‘sl4m’ seems less like of a realistic change a user would make.

Another approach we took to generate new passwords was to interpolate between two vectors in the latent space. This is similar to how GoogleBrain’s Magenta was able to

| Step | Decoded Password |
|---|---|
| 0 | noah123___ |
| 1 | noaho82___ |
| 2 | noahe123__ |
| 3 | noahua23__ |
| 4 | noahia26__ |
| 5 | noahya27__ |
| 6 | noah3a12__ |
| 7 | noahsa19__ |
| 8 | noahna11__ |
| 9 | noahra10__ |
| 10 | noahk1101_ |
| 11 | noahp1133_ |
| 12 | noahl1123_ |
| 13 | noahg1123_ |
| 14 | noaht1123_ |
| 15 | noaht1990_ |

Fig. 5. The table above shows the decoded passwords from vectors generated by interpolating between the latent encodings of ”noah123___” and ”noaht1990_”

form smooth transitions between drumbeats by interpolating between latent representations [8]. For this example we chose the passwords to be ‘noah123__’ and ‘noaht1990_.’ We took 15 steps to interpolate between the two vectors, and the decoded values are shown in figure 5.

This form of password generation seems more promising in that passwords differ in more realistic ways. Mainly, by changing the initial ”t” at the end of ”noah” and by changing the appended numbers. These changes seem more in line to the simple changes that a user might actually make.

We were also able to isolate vectors such as a unique ”l33t speak” vector that character swaps the ”3” character for ”e” characters. This was done by subtracting the latent representation for ”seen_____” from ”s33n_____.” This difference was then added to the latent representation of ”been_____” which was then decoded as ”b33n_____.” Unfortunately this vector seemed to be specific to the second and third indices of the password, because it failed to translate ”between___” to ”b3tw33n___” or similar swaps outside of the second and third indices.

Finally, we show the white-hat use case for our model as a password strength scoring engine by using the reconstruction error. In this case we define reconstruction error as the number of characters that are changed after encoding and decoding the original.

For the engine to be effective, it should give a high reconstruction error and therefore a high score to passwords that are random because these types of passwords are recommended by security experts as being the least likely to be compromised by password stuffing and brute force attacks. Meanwhile it should give a low reconstruction error and therefore a low score to passwords that are from the test set and therefore

actual instances of users passwords, all of which have actually been compromised.

To test whether this is true, random passwords were generated in two ways. The first way was done by sampling ten letters randomly from a uniform distribution over the vocabulary. Sampling from this distribution produced passwords with many unpopular characters such as curly braces. The second way was by estimating the character distribution by looking at the popularity of characters in the data set. Sampling from this distribution led to passwords with much fewer special characters.

Figure 6 shows average score errors when sampling from the test set, random characters from the char distribution, and random characters from a uniform distribution. The average reconstruction error (out of a max of ten) is calculated over a 1000 samples, and 15 samples are explicitly shown before and after encoding and decoding.

These results are promising in that the variational autoencoder behaves how we would want an engine that scores passwords to score. Passwords similar to the training set are very low at 1.71, passwords with characters sampled from a realistic distribution are scored medium at 5.976 and passwords drawn uniformly randomly (as suggested by security experts) are scored very high.

## VI. Further Work

The focus of this work was to demonstrate how variational autoencoders can be used to mimic the patterns of human behavior during password generation. And although this attempt seems successful judging by the qualitative evaluation of the network output, we cannot provide strong quantitative results to describe our performance. Future work should strive to compile and test a data set that describes how individual users' passwords change over time. If this set could be constructed, stronger conclusions could be made about the models ability to predict future password structure and content.

## VII. Conclusion

Passwords continue to rule the modern internet as the defacto authentication method. However, their weaknesses are well known. With users consistently reusing passwords from site to site with only minimal modifications being made between them, finding a the credentials to one account usually means you are not far from finding the credentials to many more. In this paper we demonstrated two uses for variational autoencoders trained on leaked plain text passwords. First, using their generative functionality to create many similar passwords from a single seed. And second, using their ability to encode and decode through a latent space as a form of anomaly detection for scoring the novelty and strength of proposed passwords.

## Acknowledgements

Thank you Professor Dr. Ditzler for the instruction this semester and Heng Liu for grading.

## References

[1] M. C. N. B. Anupam Das, Joseph Bonneau and X. Wang, "The Tangled Web of Password Reuse," 2014.

[2] P. G. K. P. G. L. M. L. M. L. B. N. C. Richard Shay, Saranga Komanduri and L. F. Cranor, "Encountering Stronger Password Requirements:User Attitudes and Behaviors," 2010.

[3] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *CoRR*, vol. abs/1312.6114, 2014.

[4] C. Doersch, "Tutorial on variational autoencoders," *ArXiv*, vol. abs/1606.05908, 2016.

[5] D. Miessler, "SecLists," https://github.com/danielmiessler/SecLists.

[6] F. Chollet, "Building Autoencoders in Keras," https://blog.keras.io/building-autoencoders-in-keras.html, 2016.

[7] N. Thurston, "VAE-Password-Prediction," https://github.com/noahthurston/VAE-Password-Prediction.

[8] A. Roberts, J. Engel, and D. Eck, Eds., *Hierarchical Variational Autoencoders for Music*, 2017. [Online]. Available: https://nips2017creativity.github.io/doc/Hierarchical˙Variational˙Autoencoders˙for˙Music

| Sampled from the Test Set | | |
|---|---|---|
| Original | Reconstruction | Error |
| gsf666____ | gsy666____ | 1 |
| Simpsons__ | nimpssns__ | 2 |
| donquijo__ | donkusgo__ | 3 |
| hubba_____ | hubba_____ | 0 |
| dorain1___ | dorain1___ | 0 |
| 110589h___ | 1105861___ | 2 |
| zp1986____ | zp1986____ | 0 |
| patone____ | patone____ | 0 |
| whymen5off | whymersr.7 | 5 |
| parfume___ | parfuml___ | 1 |
| walterervi | walteriste | 4 |
| zllemtllz_ | zllemgace_ | 4 |
| fleagle___ | fleaglo___ | 1 |
| y0a9n5c9h3 | yea9nae193 | 5 |
| ygfxBkGT__ | YKyHBbeT__ | 6 |
| | | |
| Average Error (1000 samples): | | 1.71 |

| Randomly Generated from the Estimated Character Distribution | | |
|---|---|---|
| Original | Reconstruction | Error |
| ls6sga9191 | gs6sga2101 | 3 |
| m7eoZAhmda | 1peofoleka | 7 |
| 5zlk6tae14 | 6g7k66qee7 | 7 |
| 311k4idnk5 | W11k42iret | 6 |
| 4ahnDpzeml | Yahnrdisef | 7 |
| Tz8Y3g1la2 | vg8y367321 | 8 |
| a317zo20aw | hs17lar113 | 8 |
| nie3rblr56 | nie3rais13 | 5 |
| ya91esrele | yab1essale | 3 |
| nd9hn57s3h | edshna111_ | 7 |
| 3zaVootrs7 | wgapo9eev7 | 7 |
| a3tft1aamc | aftftathtd | 6 |
| eayee0nkxa | eayeesiol1 | 5 |
| iB0iAm3d7i | vq0i63erv7 | 8 |
| k2d7e99002 | k2d7202199 | 6 |
| | | |
| Average Error (1000 samples): | | 5.976 |

| Randomly Generated from the a Uniform Character Distribution | | |
|---|---|---|
| Original | Reconstruction | Error |
| QM(dB!8c\|J | ZM9dB8Z3NQ | 7 |
| +J'mqzVli[ | xfpmf8qeV0 | 9 |
| !3F7wXyh@m | v3nVw67cfZ | 8 |
| _Ue+E\L`xh | YNe499ZQEc | 9 |
| ^@n?CH(6Gz | weFr3CXEVc | 10 |
| 46.V'@vhB$ | x6K744qEHB | 9 |
| 81$%*~HF9y | z14HL6qE1F | 9 |
| ;S#}KNKs_l | YGOnMTCNY_ | 10 |
| `6oW;2&Y@E | x6oWF6731A | 7 |
| 2kHd;n_^'+ | wkxdFatbrc | 8 |
| voLhNjhh0[ | woxLn8keV0 | 9 |
| TBk754^qOX | vQkW56ZEku | 8 |
| .bpwjA{~Nw | xKpwjZtzNc | 6 |
| _KwjZ!1d!z | XKw6q6qcBc | 8 |
| -RD=a7bmM@ | Y5Z0a973ud | 9 |
| | | |
| Average Error (1000 samples): | | 8.291 |

Fig. 6. Reconstruction error of passwords sampled from the test set, passwords generated through randomly sampling from the estimated character distribution, and passwords generated through randomly sampling from a uniform distribution.