Password Prediction:

Predicting passwords with the use of a Neural Network

Andrew Buxa

Integrated Research Component II

Prof. Louis Oliphant

April 12, 2017

**Abstract**

Passwords are often used to control access to a user’s account. These passwords are stored in a hashed format. This project seeks to train a Neural Network on these hashed passwords to recover the original password. Now, to train the Neural Network, I generated a training set of 752,000 hashed passwords. Then, I used it to predict the passwords for a test set of 3765 unique hashed passwords. The Neural Network recovered one password completely from the test data with others being off slightly.

**Introduction**

Password cracking is a problem in the field of Computer Security. It’s usually done by using a brute-force attack that tries to guess a password of a user and see if the hash it produces is identical to the actual password. It is certainly interesting because the amount of time it takes to try and guess all permutations of a password and then check their hashed value. Now, what I have done is try to create a collision (which is essentially finding a hash that matches the hash of the actual password) by running all permutations of a length-*n* password through a Neural Network to see if it can predict the actual password. It should be effective since Neural Networks are very good pattern matchers and it should see if artificial intelligence can be used to crack passwords, rather than using brute force.

**Relevant Work**

In regards to relevant work, there has not been too many people attempting to use a Neural Network to find collisions with password hashes. Some people have suggested it is possible to train a Neural Network on a dataset of cracked passwords to predict and un-cracked password. However, I noticed that some people have been using Neural Net libraries like sci-kit learn. I did not follow through with using that library because I was unable to use it.

**Project Description**

Before putting my Neural Net into code, I had to decide what library I wanted to use because sci-kit learn1 was not going to work. After looking around, my professor and I came across a library called PyBrain2, this library allowed me to easily set up a Neural Network. Now, to even use the PyBrain library, I needed to establish a dataset and training set to work with.

I started by creating a program that generates all permutations of a password of length*-n*. This would allow me to train the Neural Network on almost every possible password with a set length. The program would generate all passwords and their hexadecimal digests by running each password through the MD5 hashing function. Though, it is necessary to have test data on too, so I decided to have a random number generator that would generate a number between 0 and 6,000. If the number was above 5,900, then I would add that password to the test data file.

For example, *Figure 1* shows how the passwords are stored:

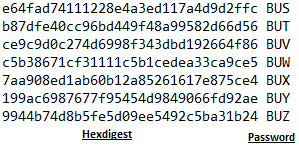


Figure 1 shows Hex digest and Character Password stored in a file.

Next, I (with the help of my professor) decided to take the hexadecimal digest and character password and convert them into binary. Since Neural Networks are pattern matchers, it is much easier to find a pattern among 1’s and 0’s rather than 26 different letters, not including capital letters and symbols. While converting the character password into binary, it is good to take note that the number conversion will cut off leading 0’s, so you will not have a binary “string” of length 24, given the 3-character password and appending each of the character binary values to another.

*Figure 2* shows the hexadecimal digest and character password converted into binary:

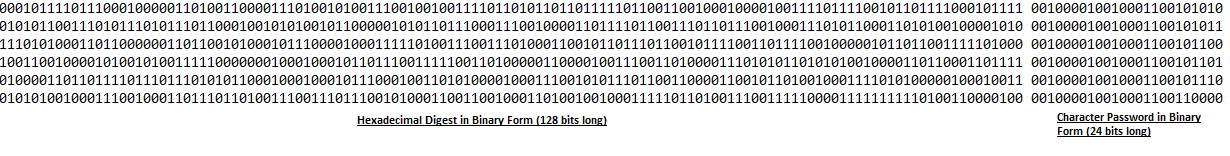


Figure 2 shows the Hexidecimal digest in binary form (128 bits long) and the Character password in binary form (24 bits long).

After all the conversions are done to both the test-data and training data, the Neural Network must be trained. To train the Neural Network, you should first specify the size of the input and output, which in this case is (128,24). The 128 will be the input and the 24 is the output, a good understanding of this would be in *Figure 2* where the hexadecimal digest is the input and the character password is the predicted output. To do this, I created an empty array and went through the hex digest and character array and added each position of the string into an element of the array, thus, giving the Neural Network its required input and output length. Once training was done, it was crucial that I added user input to the program, primarily to allow the user to control what is being done. There was a command for training, testing, checking for accuracy, and a simple exit. The most extensive command was testing because this was where all the logic for determining accuracy and organizing the predicted data took place.

For testing, I had to take in the binary converted test data that I generated earlier and then append the character password into a blank array like I did for the training phase. When that was done, I let the Neural Network run with the given data and it would output its predictions for each element in the array. The closer to 0 the predicted value is, the more the Neural Network is confident that there should be a 0 there, it is the same for 1 as well with 0.5 being the median. I then cycle iterate over the output array and append either a 0 or 1 to an empty string to create the actual binary representation of the predicted password.

Now, I need to calculate the residual values for each element in the output array and sum them together with the other sets of data to calculate the overall accuracy of the Neural Network. The lower the overall residual value is, the better the Neural Network is with its predictions. I perform this by taking each value in the output array and subtracting it by the actual character that should be there from the test data. For example, if the output for the character at position one was 0.424678, but the actual character that should be there was a 1, I subtract the predicted value by the actual value. Then, I keep track of the indices of the residual value array because I need to sort the array to decide which character I need to change to guess the password correctly. After sorting the residual value array by off-setting each value by the median 0.5 and sorting lowest to highest, I then find the original index points to know where the characters should be in order. The order in which the sorted array is organized is the values the Neural Network is least confident about (lowest) to the most confident values (highest). Then I need to iterate over the sorted array and check each character in the array to see if the actual character (from the test data because I have the actual 24-bit password) is equal to the predicted password. If it is not, I keep track of the original index of where the characters do not match. When the loop is done, I now know how many places I would have to change to correctly guess the password. I then increment an index in a score array based off the positions I need to change to get the right password.

**Results and Evaluation**

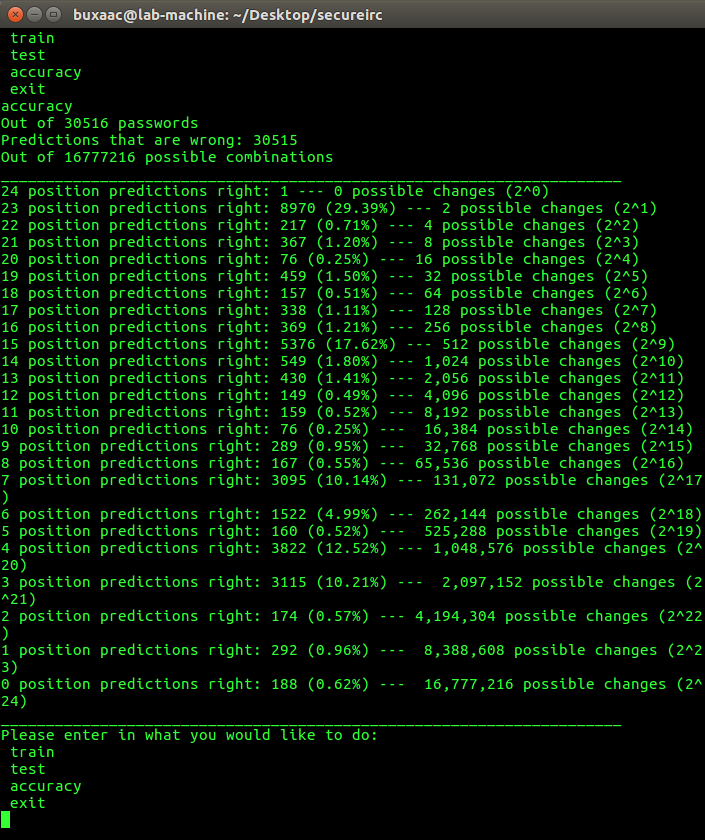
After running my Neural Network and training it multiple times to increase its “intelligence”, I was astonished by the results it gave me (*Figure 3*).  


Figure 3 shows the accuracy of the Neural Network.

As you can see in *Figure 3*, out of 30,516 test passwords, I got 30,515 wrong. But that is alright because I want to see how close the Neural Network was to guessing the correct password. By 24 position predictions right, I mean that every single position in the password was guessed correctly, so 0 bit flips need to be performed to make it correct. As you go down in the list, with 23, 1 bit flip is needed at position 1 of the predicted password to get the password correct. This means if I change 1 bit, I will get 8,970 or 29.39% of the passwords correct. That is incredible, however, going down the list you’ll see the accuracy dropping which is normal to see. Regardless, these results show that it is entirely possible and more beneficial to use a Neural Network to try and predict someone’s password rather than use the brute-force attack which will take a long time. Even if the predicted password is not correct, you only need to make a few changes, as you can see on the right of the output.

When I originally set out to put this Neural Network together, I had no intention of seeing such amazing results. Of course, I was not able to increase the length of the passwords because my account had a limited amount of disk space. None-the-less, my Neural Network does work on the same kind of level that others would in terms of predicting passwords. If I were to start today, I would make it possible to generate passwords of multiple lengths to increase the realism of guessing someone’s password, because normally you do not know the length of someone’s password.

**Conclusion and Future Work**

If I had more time, as I said in my evaluation, I would generate passwords of different lengths to increase the realism of the project. In addition, I would change some of the logic I use to reduce the total amount of time the program takes to run (excluding training which takes a while). If I had to start over, as generic of a response as this is, I would not put off the coding part of the project for a few weeks. Although, I did work on the project early in the semester, I took a little break in between to work on other school work.

I learned that it is possible to have a Neural Network predict someone’s password, or get relatively close to predicting the password. It is both frightening and fascinating that a Neural Network can be used this way.

**References**

1. http://scikit-learn.org/stable/
2. http://pybrain.org/docs/