Password generation

by Vinayaka Hegde

Submission date: 28-Apr-2021 09:31AM (UTC+0530)
Submission ID: 1572057481
File name: PW21SE01.pdf (754.09K)
Word count: 7616
Character count: 39370

CHAPTER 1
INTRODUCTION

Passwords are being used for authentication purposes for a long time. It is easy and convenient for implementing and using it. Companies need to make sure that the user's passwords are not compromised. If compromised it can be misused which affects the user's account which may lead to loss of money, comprise identity, etc.

In the past there were many attempts to hack the databases and has also happened which is concerning. The passwords are stored in hashed form in the database. If the passwords are strong enough, it will be very difficult to hack it. There are numerous ways like dictionary attack, rule-based attack, etc. to hack the passwords. These methods require expertise and are time consuming tasks. The output space is also limited.

Generative adversarial networks can be used to generate passwords that humans are likely to use. We need to train the model based on a list of real passwords that people have used. This data is available as their instances in which the compromised passwords were put online.

These GAN's without the domain knowledge will be able to understand how passwords are created and once properly trained, they will generate passwords which are likely used. This approach does not require for us to input any rules, only a list of passwords is required to train the model. Well trained models can be used to substitute or used in addition to the existing tools for password cracking which may lead to better results. Certain patterns in passwords which might not be easy to identify by humans can be understood by this model. The output space of this model is not limited and can be used to generate many passwords. In addition to its application for password cracking, it can be used to determine the strength of any password so that users make sure their password is strong is less likely to be guessed.

The model consists of two main components called generator and discriminator. Discriminator outputs the likeness of the password being created by humans. Generator tries to generate more human-like passwords and fool the discriminator.

CHAPTER 2
PROBLEM STATEMENT

Password generation is a very common security issue. Many users face the problem of choosing good passwords and try to make sure it is strong and not guessable.

Train a generative adversarial network which has 2 components namely generator and discriminator. Generator should generate password guesses to break a password scheme and the discriminator tries to prevent the generator from breaking the password scheme.

In other words, discriminator will discriminate between generated and real passwords from the input dataset and the generator tries to generate a real looking password to fool discriminator.

There should be provision to train the model based on different datasets and provision to generate the required number of passwords. There should also be provision to determine the strength of a given password based on likeness of being able to guess the password.

The proposed model can be used as a security measure where the user can check for easily crackable passwords. It can find its applications in checking password strength.

Compare this model with other tools like hashcat and check the strength of the password.

CHAPTER 3
LITERATURE REVIEW

In this chapter, we analyze the research made, and results obtained in the field of the project, by considering the various parameters and extent of the project.

3.1 Adversarial Password Cracking [1]

In this paper, a generative adversarial network is trained with a generator trying to generate guesses to break a password scheme. The author has later compared the strength of this adversarial model with rule based methods such as hashcat . Later, the passwords cracked by PassGAN are compared with results of zxcvbn, to determine if the passwords cracked by PassGAN were hard to crack.

The author suggests using a machine learning based approach for password guessing which is called PassGAN. PassGAN is a generative adversarial network where the generator will learn the input data distribution and generate a real looking password and the discriminator will learn to distinguish between real and fake passwords. They both end up playing a minimax game and optimize the value function given by Equation 1.

Equation 1 : Min Generator [maximize Disciminator] Value(D,G) = Ex ~ real(x)[log D(x)]

In this paper, the author has trained the PassGAN model on two datasets - one is RockYou and second is AshleyMadison dataset. Further, an extended version of RockYou dataset was created by including slightly modified variations of the passwords, to represent a more realistic set of passwords.

The network was trained upto 70k iterations and a plot of training loss versus iterations is represented by Figure 1.

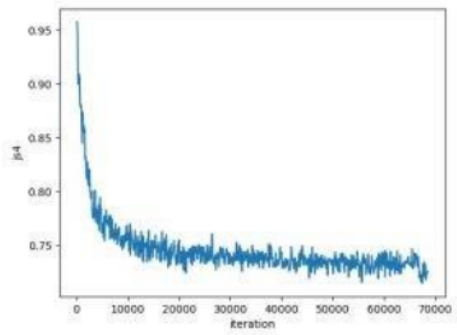

Figure 1 : Training loss v/s number of iterations

After training, the set of passwords cracked using PassGAN was evaluated using zxcvbn and the results for RockYou and extended RockYou dataset are summarized in Table 1. From this table, we can infer that the average number of guesses for extended RockYou is high since it is a much-randomized version of rockYou dataset.

Finally, the author concludes by mentioning that PassGAN can also be used to determine the strength of the password by counting the number of trials it takes before arriving at the correct password.

Data	Avg. Score	Avg. Guesses	Log Guesses
RockYou	1.5925	5.32×10^6	6.2561
Ext RockYou	1.5561	6.8×10^7	6.166

Table 1 : Average guesses for RockYou and extended RockYou datasets

3.2 Presenting New Dangers: A Deep Learning Approach to Password Cracking [2]

This paper describes how deep learning can be applied to password cracking problem. In this paper, the author argues that GANs generate high quality password guesses and can match 51 to 73 percent more passwords than the traditional rule-based methods. This shows that deep learning methods can be used for password cracking.

The paper begins by stating how carelessness of users can be exploited by malicious hackers in this online world. Next, the paper presents us with various kinds of attacks used by the current password cracking tools such as hashcat. Brute force attack generates passwords by going through a combination of characters upto a given length. Dictionary based attacks commonly match passwords from a list of passwords that are stored in a dictionary or commonly used passwords revealed from past password leaks, and then turn them into hashes.

Then, the hashed passwords are checked against the unknown hash to crack the unknown hash. Dictionary attacks mainly succeed because of the assumption that people have a tendency to choose ordinary words or passwords similar to the list of common passwords with little variation. Rule based attacks use a set of rules on top of the dictionary list. For example, there could be a rule for concatenating all passwords with some numbers since these kinds of passwords are highly probable.

The paper evaluated the performance of PassGAN by running the model on different datasets. The results prove that PassGAN can match 51 to 73 percent more passwords than the traditional rule based methods. Additionally, it was found that the best overall performance was achieved when PassGAN was combined with other techniques, specifically when PassGAN and HashCat were combined to a single model. This concludes that PassGAN was able to generate real-like passwords and when hashcat was used on these passwords, it was able to perform much better since the password list generated by PassGAN consisted of highly probable passwords.

The author concludes by stating that PassGAN can be effectively used to generate passwords that resemble real passwords since it can extract properties of passwords which could be indistinguishable to human observation. Also, PassGAN can be used without a priori knowledge of the structure of passwords. Also, PassGAN can generate passwords indefinitely whereas other tools can generate a limited number of passwords. The best overall performance was achieved when PassGAN was combined with other rule based techniques.

3.3 PassGAN: A Deep Learning Approach for Password Guessing [3]

This paper is the first attempt to apply deep learning models like PassGAN to the password guessing problem. In this paper, the author starts by stating various kinds of rule based methods available for cracking passwords and later compares the results of using these methods with

respect to that of PassGAN.

Moving on to the architecture of PassGAN, this paper uses the Wassertian GANs (IWGANs) with Adam optimizer. Various hyperparameters like batch size, number of iterations, output sequence length, size of the input vector has been stated. IWGAN is implemented using ResNets since ResNets try to continuously reduce the training error in every epoch as the number of layers increases. Residual Blocks are neural networks comprising of two 1-dimensional convolutional layers, connected with one another with RELU activation functions.

PassGAN was tested on the classic RockYou and linkedin leaked datasets. Figure (2) represents the number of unique passwords generated by PassGAN trained using the RockYou dataset. From the figure, we can infer that the number of passwords matched increased until 199,000 iterations. And after that, increasing the number of iterations reduced the number of passwords matched which implies the model started to overfit. So, after 199,000 iterations, the training was stopped.

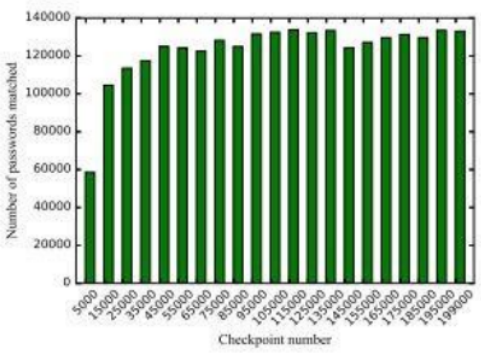

Figure 2 : Plot of passwords guesses v/s no. iterations

When PassGAN was compared with hashcat, the it was observed that PassGAN was able to match more passwords than hashcat and the most important observation was that PassGAN was able to guess passwords from a completely different dataset from the dataset it was trained with.

The paper concludes by stating major advantages of PassGAN over the rule-based approaches and in which situations rule based methods are preferred over PassGAN. It has been observed that rule-based methods were able to outperform when the number of guesses allowed to crack a password was small. However, the main disadvantage of rule-based password guessing is that

rules can generate only a finite, small set of passwords. In contrast, PassGAN was able to generate a large number of passwords. Therefore, the best strategy is to use multiple tools since it was observed that by combining the output of PassGAN with the output of the rules of Hashcat, about 50 percent more passwords could be matched.

3.4 Password Guessing using Deep Learning, 2019 [4]

The paper introduces various kinds of data breaches and the possible causes for the same. Then, it investigates how well the PassGAN model performs when trained on a large dataset. PassGAN was trained on the Russian email -based dataset and was found that PassGAN did not perform very well when trained on the dataset. After providing suitable explanations for the above-mentioned result, the paper concludes by stating some key points to be remembered while training GANs on any type of data.

The paper first introduces various kinds of data breaches using attacks like dictionary attack, brute force, rainbow attacks. The popular dictionary attacks can be described as guessing passwords by using a specialized wordlist. Brute force attack uses scripts to generate passwords systematically. But the downside is that for every increasing character, time taken by brute force to crack password, increases exponentially. Rainbow table attacks store pre-computed hash values so that time taken to crack a password significantly reduces.
Next, the paper explores various causes for data breaches. The most common is weak credentials or choosing the same passwords for multiple websites. In phishing, the attacker tries to disguise as a legitimate or trusted entity to gain access to sensitive data. Social engineering involves using some kind of manipulative tricks to get sensitive information. Ransomware is becoming increasingly popular where data of an organization is infected and restricted until a ransom amount is paid.

After the model is trained on the Russian email-based dataset, few interesting results were obtained. The most interesting result was the fact that PassGAN was able to detect passwords based on reverse translation technique. Reverse translation involves translating text from one language to other. For example, passwords where some Russian words written in English. These words may seem gibberish to English-speaking users, but they are easily recognized by Russians.

Analysing the results, the paper concluded that PassGAN did not perform very well when trained on this dataset. Table (3) represents the passwords that were not matched by PassGAN. On analysing passwords from Table (3) , it was found that since there was a lot of variation in the structure of passwords in the dataset , PassGAN could not learn a proper distribution that could represent the passwords in the dataset.

NON-MATCHED PASSWORDS

erzhikw	123ko1	trushw	m7802
pirat1501	tj987654	5555223a	lgvi2403q
tybijr123	1842825	Dania200	290292
Aleksrif	tera7	avril509a	m7zenswaq
O7kyr	countesS	menotyou0	nbemej
050613na	babachk	110764m	OOO959881
ixodes23	q12445q	home333	91298123
tapehipu	Nelson09	dbhub	sek790i26
R1der007	13028310q	2403lochs	JersecK
vorovauka	n4055071q	g6l6d6	9KABG2KU

Table 2 : list of non matched passwords

CHAPTER 4

PROJECT REQUIREMENT SPECIFICATION

4.1 Introduction

Adversarial password cracking is a method to generate passwords which could be used to determine the strength of password and also used instead of conventional methods. Complete requirements for this project are clearly specified in the following paragraphs.

4.2 Project Scope

The proposed model can be used as a security measure where system administrators can check for easily crackable passwords. It can find its applications in checking/suggesting password strength.

The model can substitute other password cracking tools for generating passwords since it does not require knowledge of the dataset. Also, the samples generated are not limited to a password space.

4.3 Product Perspective

Conventionally the password cracking tools use brute force, dictionary based and rule based generation of passwords. Creating rules requires human expertise and is time consuming. Instead GANs could be used which learns from the data and generate passwords which need not know the domain knowledge.

4.4 Product Features

- It generates password guesses which could be used in password cracking
- It can tell whether the passwords are strong or weak

4.5 User Classes and Characteristics

4.5.1 Admin

- Responsible for training the model and has access to generate passwords
- Has access to view generated passwords Check the strength of password

4.5.2 User

- Check strength of password using a user interface

4.6 Operating Environment

- This system requires python, tensorflow and keras to function.
- Presence of fast GPU ensures faster training however is not mandatory.

4.7 General Constraints, Assumptions and Dependencies

The assumptions made:

- The training dataset quality is good, and volume is high
- Fast cpu/gpu is available to train faster

Dependencies:

- The model uses tensorflow and keras inbuilt libraries for implementation.

Constraints:

- The performance of the model is constrained by the amount and quality of the dataset being used to train the model.

4.8 Risks

- Since an ML model is used, the model could overfit after training for many

iterations.

- Making changes in the model consumes a lot of time since training the model requires a large amount of time.
- ML model has high resource requirements and there is a possibility of resource overflows.

4.9 External Interface Requirements

4.9.1 User Interfaces

- Required screen formats with GUI standards for styles.

4.9.2 Software Requirements

- Python
- Tensorflow
- Keras

CHAPTER 5
SYSTEM REQUIREMENTS SPECIFICATION

5.1 Description

System requirement specification contains the description of requirements of software that needs to be fulfilled. The requirements can be functional as well as non-functional. Functional requirements indicate the functionalities that must be necessarily incorporated into the model and the nonfunctional requirements indicate the various quality constraints that must be satisfied by the system.

The following section contains the description which contains functionalities of the application, interaction of users with various components of the application. This is followed by the functional as well as non-functional requirements of the application.

5.1.1 List of functionalities supported by the application

- There must be a pre-processing module before the input is fed into the model so that only quality passwords (passwords that represent the distribution of data) must be fed into the model.
- The application enables the training of Generator and Discriminator models by updating weights after every epoch.
- There is a functionality for users to determine the strength of the input password by calculating the number of trials before arriving at the correct password.
- The application also enables admin to tune the hyperparameters of the model to achieve more efficiency.

5.1.2 Interactions between user and various other components

When the admin wishes to train the model, he/she should have the possibility to choose

the dataset they want to use. There should be an option to split the dataset to train set and test set. They should have the option to choose how many epochs to train on.

Once the model is trained, they should be able to check how many of the passwords present in the test set/dataset was the model able to generate. They should have the option customize the model later if required. They should be able to train the model using different datasets. Admin and normal users should be able to use this software to generate as many passwords required.

Software should provide an easy to use user interface for this purpose. There should be a provision to export the generated passwords which can be used according to the needs. They should be also able to check the strength of the password. Strength of password should be determined on the likeness of the password being guessed.

5.2 Functional Requirements

- Provision to train if new dataset is provided
- Interface for admin
- Provision to customize the model
- Check the strength of password
- Export list of generated passwords
- Interface for users

5.3 Non-Functional Requirements

5.3.1 Usability

The frontend / GUI must be designed so that it is easy to use, and for every action the user performs, there must be a valid response from the back end. Also, the front end must be detailed so that even a user without deep technical knowledge, must be able to appreciate and understand the overall working of the system.

5.3.2 Reliability

Since ML is used, the model must be able to provide accuracy measures to prove that it is reliable. Also, the results of the proposed model must be compared with

other existing / new models to interpret the performance of the proposed model. Also, the limitations of the model and the various conditions in which the model will outperform / underperform must be stated clearly.

5.3.3 Performance

For any testing instance, the model must provide results as fast as possible. The model must be trained on a high-speed GPU so that the training time can be minimized. While training, the output / state of the model must be saved after some number of epochs so that later, training can be resumed from intermediate epochs as well.

CHAPTER 6
SYSTEM DESIGN

6.1 Introduction

This section discusses the for 'adversarial password cracking'. This is a generative adversarial network model which needs to be trained on a list of passwords after which it will be able to generate more likely password guesses. This model can also be used to determine the strength of the password as well.

6.2 Current System

Currently there are various methods of generating password guesses like rule-based generation which is time consuming and requires expertise for creating rules. There are also GAN models, we aim to generate more likely passwords.

There are various methods to check the strength of password, this also uses some rules, checks for patterns, etc. This model once trained will be able to guess the likeliness of password being cracked. This can be used to determine the strength of the password.

6.3 Design Considerations

6.3.1 Design Goals

The design goals of our project are:

- The model must be able to generate many passwords unlike the rule-based methods where the number of passwords generated is mainly limited by the number of rules defined.
- The model must generate more realistic passwords and outperform all the current rule-based systems.

The principles of design are:

- The software should be designed in such a way that it accommodates and adjusts to the changes done in different phases of software development.
- The design process should not reinvent the wheel. So, we plan to use inbuilt keras modules whenever possible for deploying machine learning models.

The guidelines for project development are:

- The project must be developed using agile methodology so that based on the outcomes, we can make necessary changes to the project at regular intervals.
- Comprehensive validation/testing must be conducted against the specified metrics so that it is clear if deliverables meet the requirements.
- Extensive documentation and maintenance of the code of the project using version control systems.

6.3.2 Architecture Choices

Some of the alternate architecture choices that can be considered are:

Using autoencoders (or any other generative models) instead of GANs to generate passwords as both are unsupervised learning methods.

Pros of using autoencoders over GANs:

- Autoencoders project the input data to a lower dimension and transform it back to the same shape while GANs do not lower the dimension of the data.

Cons of using autoencoders over GANs:

- Autoencoders are used to restructure the input data and cannot generate new samples while GANs learn the distribution of the input data to generate new samples of input data.
- Autoencoders introduce a deterministic bias whereas GANs do not introduce any bias.

We can also use supervised models instead of GANs to generate passwords.

Pros of using Supervised models over GANs:

- Training a GAN requires finding a Nash Equilibrium. Nash equilibrium is not guaranteed, and sometimes model parameters may not converge if nash equilibrium does not exist.

Cons of using Supervised models over GANs:

- The complexity of GANs is less when compared to supervised learning models and it is very difficult to get labels for leaked passwords dataset.

By considering the above-mentioned points, using GANs is the best architectural choice because it can generate passwords by learning the input distribution and also does not require a labelled dataset.

6.4 High Level System Design

Following are the various kinds of components of the project:

- Model component

  This is the core component that is responsible for implementing the GAN model. This is divided into subcomponents called Generator and Discriminator.

- Generator component

  This component takes random distribution as input and generates passwords after the completion of training. This has methods called generate () and updateWeight() and it interacts with the Discriminator component.

- Discriminator component

This component takes the output of the Generator component and real samples from input distribution and classifies passwords as real or fake. This has methods called classify () and updateWeight().

- Client / User component

  This component uses functionalities provided by the model component for checking the strength of password input by the user.

- Admin Component

  This component is mainly responsible for providing input data, managing the training process (like customizing the hyperparameters) and verifying the generated passwords. Admin component uses functionalities provided by model component for training and evaluation.

Storage and other information

- The model is trained on a Kaggle kernel with GPUs since the amount of data is huge and it also reduces the training time. Therefore, the input dataset is also uploaded to the Kaggle kernel.

The logical data flow of the project is represented by the below diagram:

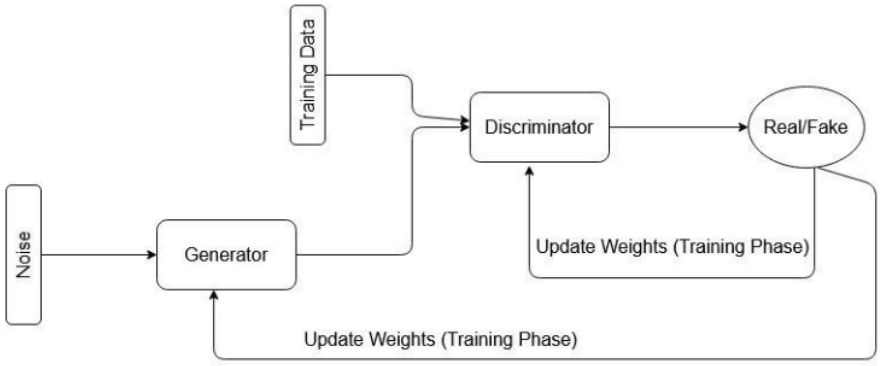

Figure 3: Logical workflow of the project

6.5 Low level design

The overall project is broken down into 5 classes:

- User: User module has functionalities to check password strength and generate password.
- Preprocessing: This class provides the data and also deals with pre-processing the same.
- GAN Model: This is a generalization of Generator and Discriminator classes.
- Generator: This consists of methods for training the generator model.
- Discriminator: This consists of methods for training the discriminator model.

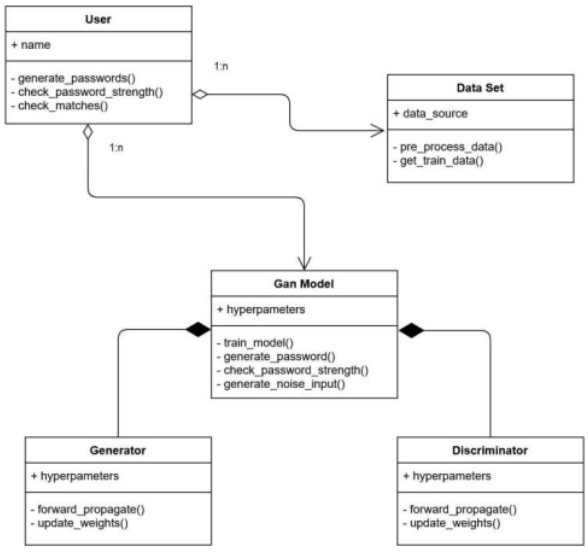

Figure 4: Master class diagram for the high-level design of the project

6.6 User Interface

The following indicates the main functionalities to be implemented as a part of User Interface:

- Provision to process the data for training

- UI for training data
- UI for generation of passwords
- UI for checking the strength of passwords

CHAPTER 7

IMPLEMENTATION AND PSEUDOCODE

7.1 Preprocessing Modules

Password is a string which is a sequence of characters. However, we need to pass numbers to neural networks. We need to encode the string to a sequence of numbers. We have created a separate file in which each valid character (all printable characters except space, tabs) is mapped to an integer.

This mapping is stored using a dictionary (encoding Dictionary) in the program with character as key and integer value as the value. Another dictionary (decoding Dictionary) is maintained to decode the output from the model. It stores the integer as the key and the character as the value.

7.1.1 Create Dictionary

This method takes as input the dict.txt file that has mapping of every character to it's respective integer value. First, the file is opened, read and split by newline as the delimiter. The, every line in the list is split by considering space as the delimiter. Now, the first value gives the character and the second value is the corresponding integer mapping. Later, two dictionaries are created – one for encoding and other for decoding.

The encoding dictionary gives the integer value when the character is supplied. So, the character is made as the key and the corresponding integer mapping forms the value. The decoding dictionary gives the character when the integer is supplied. In this case, the integer is made as the key and the decoded character forms the value. Finally, the key value pairs are appended to the dictionaries.

```
def createDictonary(fileUrl):
    file = open(fileUrl,'r')
    reader = file.read()
    reader = reader.split("\n")
    for r in(reader):
        row = r.split(" ")
        newKey = row[0]
        newValue = int(row[1])
        encodingDictonary[newKey] = newValue
        decodingDictonary[newValue] = newKey
        listOfEncodedValues.append(newValue)
    file.close()
```

7.1.2 Encoding the passwords

The input passwords must be encoded to a list of integers for the model to train. To achieve this, a function Encoding String is defined that takes a password in string format as the input. First, the length of the password is computed. If the password is less than the maximum length allowed, then the remaining values in the output list must be padded with a character called as padding_char.

So, first the string is iterated until the length of the string and the corresponding encoding of the string is appended to the output list. Then, from the length of the string until the maximum length allowed, the output list is padded with the padding_char. Finally, the output list is returned which forms the integer encoding of the input password.

```
def encodeString(inputString):
    outputList = []
    lengthOfString = len(inputString)
    for c in inputString:
        outputList.append(encodingDictonary[c])
    for i in range(lengthOfString,maxLength+1):
        outputList.append(paddingChar)
    return outputList
```

7.1.3 Decoding the integer list

After the passwords are generated from the model, the generated list of integers must be decoded into passwords in human-readable format. To achieve this, decodeList function is defined that takes the encoded list as the input. A variable decodedString is declared and initialized with a NULL value.

Then, the input string is iterated till the padding character and in every iteration, the current character is appended to decodedString. Once the current character is equal to the padding character, we break out of the loop and the decodedString is returned which forms the decoded representation of the integer list.

```
def decodeList(inputList):
    decodedString = ""
    for i in inputList:
        if(i == paddingChar):
            break
        decodedString = decodedString + decodingDictonary[i]
    return decodedString
```

7.1.4 Creating the training data

Training data is the list of encoded passwords of the input dataset that is fed as the real dataset. To arrive at the dataset, the input dataset file is opened and split by considering new line as the delimiter.

For every password in the list, if the password contains multiple words separated by white spaces, then both the words are combined into a single word that forms the password. Every password is encoded using the encodeString function and the corresponding list is appended to the output list. Since the model needs the input in the form of an array, the output list is converted to a numpy array using the array method of numpy.

```
#SPECIFY FILE PATH
createDictonary("dict.txt")

train_data= []
file = open("valid_passwords.txt",'r')
reader = file.read()
reader = reader.split("\n")
for password in reader:
  if ' ' in password:
      li=password.split(' ')
      password=li[0]+li[1]
  train_data.append(encodeString(password))

train_data=np.array(train_data)
```

7.2 Generator Model

Generator takes a low dimension noise vector as an input and outputs a higher dimension vector. This output is then decoded to a string(password). Generator has four dense layers and has tanh as the activation function in the final layer.

While training the generator, the number of epochs is taken as 1000. Then, the TensorFlow modules are imported, followed by importing Adam optimizer module. From layers module of keras, Sequential, Dense, Relu modules are imported. Then, Input module, Mode modules are imported. Model_from_json module is used to save/load the model after training.

First, the training data is normalized and mapped to a range from -1 to 1 using a range mapping formula. Then, from the training data train_size and input_size is extracted. Then, noise dimensions and batch_size parameters are defined. steps_per_epoch is computed by dividing training size by batch size. And, adam optimizer is initialized with learning rate as 0.02.

```
epochs = 10000
import tensorflow as tf
from keras.optimizers import Adam
import numpy as np
from keras.models import Sequential
from keras.layers import Dense,ReLU
from keras.layers.advanced_activations import LeakyReLU
from keras.layers import Input
from keras.models import Model
from keras.models import model_from_json

from keras.layers import Activation
from keras import backend as K
x_train = train_data
x_train = (x_train.astype(np.float32) - 165) / 65
```

The create_generator method is used to define the generator model with number of layers, number of neurons in each layer, activation function and other hyperparameters. Initially, a keras sequential model is created. To that, a fully connected layer is added with 10 neurons. Followed by that, we have a fully connected dense layer with 20 neurons. Then, there is a Dense layer with 40 neurons, and this forms the output layer with tanh as the activation function. The activation function is chosen as tanh because the generator must generate continuous values so that the discriminator can use these generated values to train. Finally, the generator model is returned to the calling function.

```
def create_generator():
    generator = Sequential()
    generator.add(Dense(10, input_dim=noise_dim))
    generator.add(Dense(20))
    generator.add(Dense(40))
    generator.add(Dense(input_size, activation='tanh'))
    generator.compile(loss='binary_crossentropy', optimizer=optimizer)
    return generator
```

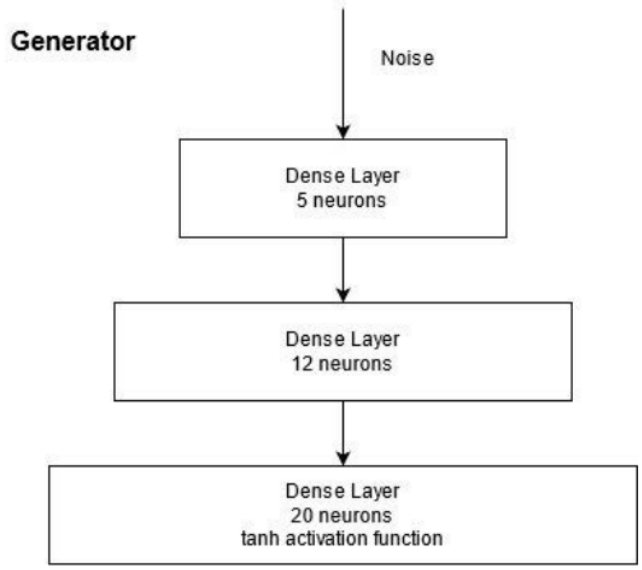

Figure 5 : Layers of the Generator model

7.3 Discriminator Model

Discriminator takes an encoded vector as an input and outputs whether the input is real or fake (generated by generator). It takes a high dimension vector as input and outputs a single value. Discriminator has 4 dense layers and has sigmoid as the activation function in the final layer.

Discriminator has a keras sequential layer as the input layer. Then, it has a sequential fully connected dense layer with 40 neurons. Followed by that, there is a Dense layer of 20 neurons. And the output layer is a dense layer with a single neuron that uses the sigmoid activation function. The optimizer used is Adam and the loss function is binary cross entropy. The loss function is chosen according to the activation function.

Since sigmoid is a classification function, binary cross entropy as loss function would perform the best. Also, the output layer has a classification activation function since the output generated by the discriminator has to be classified between real or generated. If the output is a value between 0 and 0.5, it is considered as a real password. If it is between 0.5 to 1, it is a generated password.

```
def create_discriminator():
    discriminator = Sequential()
    discriminator.add(Dense(40, input_dim=input_size))
    discriminator.add(Dense(20))
    discriminator.add(Dense(1, activation='sigmoid'))
    discriminator.compile(loss='binary_crossentropy', optimizer=optimizer)
    return discriminator
```

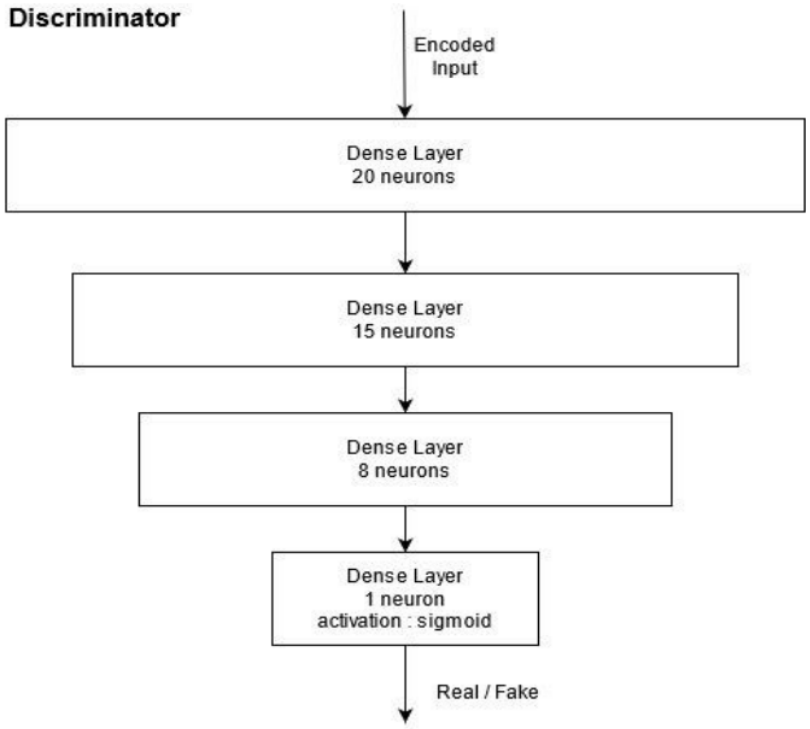

Figure 6 : Layers of the Discriminator model

7.4 GAN Module

In this section, the generator and discriminator functions are invoked and are combined to form a GAN model. Then, the model is trained by specifying the number of epochs and batches.

The create_discriminator is invoked that returns a discriminator model. Similarly, the create_generator function creates and returns a generator model. Next, discriminator is made as untrainable because during the back propagation of gradient when generator is trained, discriminator must not be trained again. A tensor gan_input is defined that takes a placeholder of noise_dimension shape.

Also, a tensor fake_image is defined that invokes the generator model and passes gan_input tensor as the input parameter. A tensor gan_output is defined that contains the output of the discriminator. Then, both the tensors are combined using the Model function that takes gan_input and gan_output as parameters. Finally, the model is compiled using binary_crossentropy loss function and Adam optimizer.

```
discriminator = create_discriminator()
generator = create_generator()
discriminator.trainable = False

gan_input = Input(shape=(noise_dim,))
fake_image = generator(gan_input)

gan_output = discriminator(fake_image)

gan = Model(gan_input, gan_output)
gan.compile(loss='binary_crossentropy', optimizer=optimizer)
```

7.5 Training Module

In the module, actual training of the generator and discriminator modules take place. For every epoch, steps_per_epoch is computed which forms the number of batches. And for every batch, batch_size number of instances are made as the noise vector. The noise vector is obtained by generating random normal samples between 0 and 1. The noise vector is fed as input to the predict function of generator which outputs the list of samples generated. Next, batch_size number of instances are picked from the training dataset and these samples form the real dataset. Both real and generated datasets are combined using concatenate method of numpy and the combined vector is fed as input to the discriminator.

To generate the label vector, first a zero vector of size twice of batch_size is created. Then, the last batch_size number of instances are made as one that results in a vector with instances corresponding to real data labelled as one and instances corresponding to generated data are labelled as zero. Next, train_on_batch function is used to train the discriminator, which has two parameters - the combined vector and the combined label vector. This training function outputs a loss after training that forms the discriminator_loss.

The labels for the generator must be a vector of all ones that indicates to the generator that the random noise vector gives as the input is the real data for the generator. So, a numpy vector of batch_size is created and the numpy vector as well as the noise vector is fed as the input to train_on_batch function for the GAN model. Since the discriminator had been set to untrainable, this trains only the generator. After every epoch, the generator and discriminator losses are printed to give an idea of the accuracy of the model after every epoch.

```
for epoch in range(epochs):
    for batch in range(steps_per_epoch):
        noise = np.random.normal(0, 1, size=(batch_size, noise_dim))
        fake_x = generator.predict(noise)
        real_x = x_train[np.random.randint(0, x_train.shape[0], size=batch_size)]

        x = np.concatenate((real_x, fake_x))
        disc_y = np.zeros(2*batch_size)
        disc_y[:batch_size] = 1
        d_loss = discriminator.train_on_batch(x, disc_y)
        y_gen = np.ones(batch_size)

        g_loss = gan.train_on_batch(noise, y_gen)
    print("-------------------------------------------------\n")
    print(f'Epoch: {epoch} \t Discriminator Loss: {d_loss} \t\t Generator Loss: {g_loss}')
```

7.6 Generating Passwords

This is the final section of the implementation of the model where the model is tested by generating passwords using the generator and if the password generated matches with the distribution of the input dataset, then the model was successful in training and learning the distribution of the input dataset.

First, a function round_off is defined that takes as input the integer encodings and rounds them off to the nearest value. This is needed because passwords generated could possibly have

decimal places and to decode the, back to their character encodings, rounding off is necessary. Further, a map_range function is defined that maps the passwords to a specific range. This range is same as the range chosen while creating the mapping dictionary. The map_range is required because the passwords generated by the generator are not limited to a specific range.

```
def round_off(ele):
    return round(ele)

func = np.vectorize(round_off)

def map_range(ele):
    if(ele >= 121 and ele <= 129):
        return ele + random.randint(9,21)
    elif(ele >= 151 and ele <= 159):
        return ele + random.randint(9,21)
    elif(ele >= 181 and ele <= 189):
        return ele + random.randint(9,31)
    elif(ele >= 221 and ele <= 229):
        return 230
    else:
        return ele

foo = np.vectorize(map_range)
```

The generate_passwords function takes as input the random noise vector and prints the generated passwords. First, the predict method of generator model is used to generated passwords. Then, the generated passwords are mapped from a range of -1,1 to the initially chosen dictionary range. For every password in this list of generated passwords, deocodeList method is applied to convert list of integer representations to the corresponding passwords in the text format.

```
def generate_passwords(noise):
    generated_passwords = generator.predict(noise)
    generated_passwords = 65*(generated_passwords.astype(np.float32)) + 165
    for i in range(no_pwds):
        generated_pwd = func(generated_passwords[i])
        final_generated_pwd = foo(generated_pwd)
        print(decodeList(final_generated_pwd))

noise = np.random.normal(0, 1, size=(no_pwds, noise_dim))
generate_passwords(noise)
```

7.7 Saving and Loading the model

7.7.1 Saving the model

The trained model must be saved so that we can load the model in the future without training it again. To achieve this, the models are first converted to json file using to_json method. Later, a new json file is created and the json is written to this file. Finally, the model weights are saved in a .h5 file format using save_weights method.

```
generator_json = generator.to_json()
with open("generator_model.json", "w") as json_file:
    json_file.write(generator_json)
generator.save_weights("generator_weights.h5")

gan_json = gan.to_json()
with open("gan_model.json", "w") as json_file:
    json_file.write(gan_json)
gan.save_weights("gan_weights.h5")

discriminator_json = discriminator.to_json()
with open("discriminator_model.json", "w") as json_file:
    json_file.write(discriminator_json)
discriminator.save_weights("discriminator_weights.h5")
```

7.7.2 Loading the model

Before loading the model, the optimizer has to be initialized and mode_from_json has to be imported from model module of keras. The saved json file is opened in read mode and all the contents are read into a variable. The, the file is closed. The model is constructed from the json using model_from_json method. Then, the weights are loaded from the .h5 file using the load_weights method. Finally, the model is compiled by specifying the loss and the optimizer as the parameters to the compile method. This completes the loading process and now the model can be directly used without the need for re-training it.

```
json_file = open('../input/gan-model/generator_model.json', 'r')
loaded_model_json = json_file.read()
json_file.close()
generator = model_from_json(loaded_model_json)
generator.load_weights("../input/gan-model/generator_weights.h5")
generator.compile(loss='binary_crossentropy', optimizer=optimizer)

json_file = open('../input/gan-model/gan_model.json', 'r')
loaded_model_json = json_file.read()
json_file.close()
gan = model_from_json(loaded_model_json)
gan.load_weights("../input/gan-model/gan_weights.h5")
gan.compile(loss='binary_crossentropy', optimizer=optimizer)

json_file = open('../input/gan-model/discriminator_model.json', 'r')
loaded_model_json = json_file.read()
json_file.close()
discriminator = model_from_json(loaded_model_json)
discriminator.load_weights("../input/gan-model/discriminator_weights.h5")
discriminator.compile(loss='binary_crossentropy', optimizer=optimizer)
```

7.8 Testing the models

7.8.1 Testing the Generator

The generated model is verified by checking if the passwords generated by the generator matches the input dataset. If it does, then the generator has successfully learnt the patterns in the distribution of the dataset. And, when trained with some other dataset with a different distribution, the generator is likely to learn the distribution of the new dataset as well.

7.8.2 Testing the Discriminator

Discriminator is tested based on the ability to discriminate between real and fake passwords. To test the discriminator, a list of passwords with equal number of real and generated passwords are fed as input to the discriminator and the predict method is used to get the discriminator predictions. If the discriminator outputs a value between 0 and 0.5 for fake passwords and values between 0.5 and 1 for real passwords then the discriminator is said to be trained as it's able to distinguish between real and fake passwords by observing the difference in their patterns.

7.8.3 Testing the combined model

After testing the generator and discriminator models individually, both the models are combined and are subjected to testing. In this process, first a train_list is created and is populated with real passwords. Similarly, a generated_list is created and is populated with generated passwords.

The train_list is iterated and for every password in the list, the password is encoded and is saved to a new list. Similarly, for generated_list, every password is encoded and is saved to a new list. Now, both these lists are combined to a list and it's fed as input to the predict function of discriminator.

For the model to be trained, all the numbers in the output of the predict function must be similar. That is, of the outputs are indistinguishable, then the discriminator has failed to separate the real passwords from the fake generated passwords. This indicates that the generator is successful in generating real-like passwords that fooled the discriminator. Hence, the combined model is said to be trained and can be used to generated new passwords that match the distribution of the dataset.

```
train_list = ['michael','ashley','qwerty','batman','password']
generated_list = ['kuroOB7;~','fuDkmCG','kqpopF2=^','jppomu0\\','htumBDS/~}','lqqqwH<1']
encoded_list = []
pwd_list = []

for i in train_list :
    encoded_list.append(encodeString(i))

for i in generated_list :
    encoded_list.append(encodeString(i))

classified_pwds = discriminator.predict(encoded_list)
print(classified_pwds)
```

CHAPTER 8

EXPERIMENTAL RESULTS AND DISCUSSION

This section discusses in detail the implementation of the model and the results obtained. For training the model, rockyou dataset is used. RockYou contains about 32 million passwords obtained from popular data breaches. Considering the training time and processing power required, only topmost 10k passwords from RockYou was used for training. Since ML models cannot work with characters, every password in the input dataset had to be encoded into a list of integers before being fed to the model.

To achieve this, a dictionary of all ascii characters was created that mapped a character to a corresponding integer. Using this dictionary, all passwords from input were encoded to lists and were stored in a numpy array. This completes the preprocessing part of the implementation. Generator was implemented using multiple Dense Sequential layers with Adam optimizer and tanh activation function used at the output layer.

Discriminator was designed similarly using multiple Dense layers with Adam Optimizer. And the output layer had one neuron for spitting out a probability of password being real or fake. Since this becomes a classification problem, sigmoid activation function was used at the output layer with binary cross entropy as the loss function.

Both the models were combined into a single model and discriminator was made untrainable so that when the model back propagates the gradient, only generator is trained. While training the model, at every epoch, the input dataset is split into multiple batches and in every batch, fake samples are the output of passwords generated by the generator and real samples from the passwords picked randomly from the input dataset.

All the fake samples are labelled as zero and all real samples are labelled as one and are fed to discriminator. Then, discriminator is trained using the combination of passwords and labels and finally the gan model is trained using a random noise vector, that trains the generator.

Discriminator is excluded from training because it was made untrainable at the beginning of the training process. This configuration was trained for 100 epochs with batch size as 16. It was found that both generator and discriminator losses started at 1.1 and later reduced to around 0.2. Also, it was found that when after every epoch, when generator loss decreases, the discriminator loss increases as expected because both generator and

discriminator form an adversarial system. After completion of training, model structure and weights are saved in a json file which can be loaded later, and the model can be loaded and used without the need for training. For evaluating the model performance, a random noise vector is fed as input to the generator.

For testing the model, a random noise vector is given as input to the generator and it generates a list of passwords. For every password in the list, the generated integer encoding is first rounded off followed by a mapping to a specific range. Later, all the passwords are decoded from their integer representation to a sequence of characters. And this final list of passwords forms the passwords generated by the model after learning the input distribution. As mentioned above, discriminator is used to discriminate between real and generated passwords.

To test if discriminator has learnt to do this, equal number of real and generated passwords were combined into a list without any labels. The list was fed as input to the discriminator so that it gives out a single value for every password. It was found that among generated and real passwords, the value given out by discriminator was nearly the same and all values were around 0.4.

This indicates the discriminator is not able to distinguish between real and fake passwords and generator was successful in generating real like passwords. Therefore, it has been verified that both discriminator and generator are trained completely and can now be used with different input data sets to learn their password distribution.

To verify if the model works as expected, the trained model was further subjected to unit testing and performance testing. In unit testing, all the methods were tested using unittest module in python. This helped in the correction of a bug in the encoding function. The bug was that the actual password length with which the model trains was one more than the expected password length.

For performance testing, the training time for various datasets were compared using a plot. Figure 7 shows the time taken against the input size. Clearly, as the input password size increases, the time taken had increased linearly. The information from this plot when combined with the GPU speed gives us a clear picture of the various sizes of input that can be trained using the model. The above discussed testing methodologies have made the model more robust and bug free.

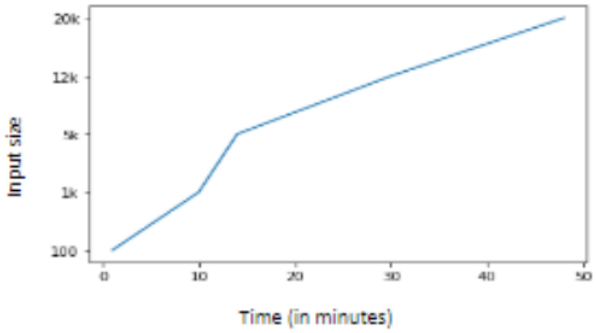

Figure 7: Performance testing of the model

CHAPTER 9

10 CONCLUSION AND FUTURE WORK

In this paper, a password generative technique based on GAN was implemented, trained, and verified using various testing methodologies. The main advantage of this model when compared to other rule-based techniques is that it does not rely on any additional information or assumptions on the input dataset. The performance of the model was evaluated using a twostep process.

Generator was evaluated by allowing it to generate a set of passwords and these were matched against the passwords from the input data set and was found that the generated passwords matched the distribution of the input passwords. Discriminator was evaluated by passing a combined list of generated and real passwords as the input and it was found that discriminator failed to distinguish generated passwords from the real ones.

The only disadvantage is that when compared with the other methods, the implemented model requires to output many passwords to start generating meaningful passwords. But this cost would be negligible when trained on a large dataset. In the future, the implemented model can be combined with the existing methods like HashCat to further increase the number of passwords generated.

First, the input dataset can be passed to HashCat that would crack most of the easy passwords and the passwords which could not be cracked by HashCat can be fed as input to the GAN model. Also, other alternative models like RNNs or Autoencoders could be implemented and compared with GAN. Finally, the existing model could be trained with other kinds of datasets like LinkedIn leaked or Ashley Madison datasets and verify if the model is able to learn the distribution when trained with these data sets.

In the first phase of the capstone project, after understanding the importance and use case of password cracking, a detailed literature survey was conducted. From the literature survey, it was evident that machine learning models could be applied for generating password guesses. On researching more about PassGAN, it was found that PassGAN could significantly outperform the current rule-based methods and the best results are achieved when PassGAN is combined with other techniques. Also, in one of the papers, checking password strength using PassGAN was

suggested as future work.

Next, the project requirements document was designed that consisted of project scope, features, constraints / dependencies / assumptions / risks involved in the project development.
Later, the system requirements document was designed that consisted of details about functionalities of various components of the system, their interaction with the user and also the functional and nonfunctional requirements of the application.

After designing the project and system requirements, a high-level design document was designed that indicated the overall design which also included a master class diagram for the system to be implemented. Suggestions given at the end of every review was considered and suitable changes were incorporated accordingly. To conclude, in the first phase we got a clear understanding of scope of project, background work done, functionalities to be implemented, and got an idea about the expected results after the project is implemented.

Password generation

ORIGINALITY REPORT

3%	1%	2%	0%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	Briland Hitaj, Paolo Gasti, Giuseppe Ateniese, Fernando Perez-Cruz. "Chapter 11 PassGAN: A Deep Learning Approach for Password Guessing", Springer Science and Business Media LLC, 2019 Publication	1%
2	www.cs.tufts.edu Internet Source	<1%
3	Vernit Garg, Laxmi Ahuja. "Password Guessing Using Deep Learning", 2019 2nd International Conference on Power Energy, Environment and Intelligent Control (PEEIC), 2019 Publication	<1%
4	"Applied Cryptography and Network Security", Springer Science and Business Media LLC, 2019 Publication	<1%
5	Submitted to Trafford College Group Student Paper	<1%
6	export.arxiv.org Internet Source	<1%

7	Submitted to University of Wales Swansea Student Paper	<1 %
8	www.autosar.org Internet Source	<1 %
9	arxiv.org Internet Source	<1 %
10	"Machine Learning and Knowledge Discovery in Databases", Springer Science and Business Media LLC, 2021 Publication	<1 %
11	Submitted to Barnet and Southgate College Student Paper	<1 %
12	senior.ceng.metu.edu.tr Internet Source	<1 %

Exclude quotes	On	Exclude matches	< 5 words
Exclude bibliography	On		