CSC 8228 - Privacy-Aware Computing

Fall-2022 – Project Report

Password GAN

Team Members
Shilpa Batthineni (002670546)
Pratyush Reddy Gaggenapalli (002676126)
Sairaj Indroju (002672402)
Sai Krishna Reddy Kandhadi (002678661)

Abstract

Unstructured data learning useful representations is one of the main problems and a driving force of contemporary data-driven methods. Deep learning has shown the numerous benefits of understanding and using these representations.

We provide a deep generative model representation learning method for password guessing in this project. We demonstrate that an abstract password representation offers compelling and adaptable characteristics that open up new avenues in the heavily researched but still active password-guessing topic. With these qualities, new password-generating methods can be created that are neither feasible nor useful when using the current probabilistic and non-probabilistic approaches. Based on these characteristics, we present (1) a general framework for conditional password guessing that can produce passwords with any bias and (2) a framework inspired by expectation maximization that allows the estimated password distribution to be dynamically adjusted to match the distribution of the attacked password set.

Password Guessing Methods

By continually putting different candidate passwords to the test, the attacker uses a password-guessing attack to try to discover the password of one or more users. HashCat and John the Ripper (JTR) are two well-known password-guessing programs today. Both programs employ a variety of password-guessing techniques, such as exhaustive brute-force attempts, dictionary-based attacks, and rule-based attacks. which creates password guesses using word changes taken from the dictionary.

These programs are capable of performing straightforward dictionary assaults as well as expanding password dictionaries utilizing password-generating rules like word concatenation (such as "password123456") and leet speak (such as "password" becoming "p4s5w0rd"). Although these principles are effective in the real world, expanding them to simulate additional passwords is a time-consuming operation that calls for specialized knowledge.

Motivation

A machine learning approach to password cracking It uses neural networks and GAN (generative adversarial learning) to guess passwords based on a given dataset. Could be used to directly guess user passwords, or applied to evaluate password strength.

How is this different from old methodologies

By using dictionary words as a starting point, the top password-guessing programs, such as HashCat and JackTheRipper, attack the "human component" that is frequently present in user-generated passwords. Strings of random characters are now more prevalent (e.g., qJnhUGz8]pBl) and nearly hard for current tools to guess thanks to password banks and Google's suggested password feature. Because it bases its guesses on the millions of passwords it practiced on, machine learning models have a greater chance of cracking "qJnhUGz8]pBl". It will improve the likelihood of a correct guess if that dataset happens to be predominately composed of passwords supplied by Google.

Generative Adversarial Networks

Using Generative Adversarial Networks, a novel method for generating password guesses (GANs) A notable development in deep learning is the use of generative adversarial networks (GANs). The generating deep neural network G and the discriminative deep neural network D make up a GAN.

Generative, adversarial, networks is referred to as GAN. The final point is the most straightforward: networks. GANs are constructed using neural networks, typically deep neural networks. A GAN begins with an input layer with a specific number of parallel input neurons (one for each number represented by the input data), some hidden layers, and an output layer. These layers are connected in a directed graph and were initially trained using a variation of the gradient-descent backpropagation algorithm.

The term "generative" follows, which refers to the goal of this group of algorithms. They create data instead of consuming it. More specifically, the information that these algorithms generate is new information that belongs to the same 'class' as the original data that produced it. Data are generated from other data using a mechanism discussed later; this process is not spontaneous.

The word antagonistic, the most enigmatic in the acronym, illustrates how generation happens by describing a contest between two enemies. Neural networks are the GAN's enemies in this scenario. As a result, a GAN tries to produce new data via networks that are purposefully opposed to one another to accomplish this goal.

A GAN is always composed of two neural (often deep) networks. The first sometimes referred to as the discriminator, is trained to tell a collection of data apart from pure noise. For instance, the input data could consist of a variety of images unrelated to flowers as well as a large number of images of flowers. Although each image may not have a clear caption, it is clear which images are part of the collection of flowers and which are not.

The network can then be trained to distinguish between flowers and other objects, as well as between actual photographs and images made from random pixels. This first "discriminator" part of the GAN is a regular network that has been taught to categorize items. If we want to create flower images, for example, we would enter a collection of flower photos, and the result would be a yes/no flag.

The other network is the generator: this produces as output the kind of data the discriminator is trained to identify. To achieve this output, the generator uses a random input. Initially this will produce a random output, but the generator is trained to backpropagate the information, whether its output is similar to the desired data (e.g., photos of flowers).

To that end, the generator's predictions are fed into the discriminator. The latter is trained to recognize genuine flowers (in this example), so if the generator can counterfeit a flower sufficiently well to trick the discriminator, then our GAN can produce fake photos of flowers that a well-trained observer (the discriminator) will take for the genuine article.

Project Dataset

RockYou: RockYou was a company that developed widgets for MySpace and implemented applications for various social networks and Facebook. Since 2014, it has engaged primarily in purchasing rights to classic video games; it incorporates in-game ads and re-distributes the games. In December 2009, RockYou experienced a data breach resulting in the exposure of over 32 million user accounts. This resulted from storing user data in an unencrypted database and not patching a ten-year-old SQL vulnerability. (including user passwords in plain text instead of using a cryptographic hash)

Link: http://downloads.skullsecurity.org/passwords/rockyou.txt.bz2

The RockYou dataset is imported and encoded with Latin-1 to read all special characters from the text file.

```
path_to_file ="rockyou-train.txt"
text = open(path_to_file, 'rb').read().decode(encoding='latin-1')
print ("The dataset has", len(text), "characters in total.")
```

```
The dataset has 172941082 characters in total.
```

Proposed Methodology

It's an unsupervised machine learning method that makes two neural networks compete against each other (their "adversary"). The discriminator attempts to determine which inputs are from the original data, and which are from the generator.

The generator imitates the target data in an effort to trick the discriminator. G's objective is to generate "incorrect" samples from the underlying probability distribution Pr(x) that are recognized by D, given an input dataset I = x1; x2;::: ; xn. D wants to learn how to tell the difference between genuine samples coming from I and false samples coming from G.

The 'Discriminator' network evaluates every piece of data for authenticity and decides if it's real and belongs to the actual training data set or a fake. This trains both networks, one to create better fakes, and the other gets better at determining if the data is real or not.

Implementation:

Unique alphabet/character values are taken from password dataset

In [4]:
```
unique=set()
for val in text:
  unique.add(val)
```

Indexing the above characters using the below code block -

```
data_vocab={}
index_vocab={}
i=0
for val in unique:
  data_vocab[val]=i
  index_vocab[i]=val
  i=i+1
```

Initialize the batch size and split the dataset with “\n” to read all the passwords and then iterate over the generated passwords and use the above-generated indexes of unique characters and create a tensor of each letter in a password.

[47]
```
batch_size = 32
temp_data=text.split("\n")
```

[48]
```
data=[]
count=0
for val in temp_data:
  if len(val)==8:
    temp=[]
    for i in val:
      temp.append(data_vocab[i])
    data.append((torch.tensor(temp),torch.tensor(0)))
    count=count+1
```

We begin the project by importing the required libraries

In [6]:

```
import torch
from torch import nn

import math
import matplotlib.pyplot as plt
```

Here, torch is used to import the PyTorch library. In order to build up the neural networks in a less verbose manner, we additionally import nn. The Matplotlib charting tools are then imported as normal along with math to get the value of the pi constant.

Implementing the Discriminator:

A model called the discriminator has a one-dimensional output and a two-dimensional input. It will be given a sample from the generator or the real training data and will then estimate the likelihood that the sample comes from the real training data. How to build a discriminator is demonstrated in the code below

In [10]:

```
class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(8, 256),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(64, 1),
            nn.Sigmoid(),
        )

    def forward(self, x):
        output = self.model(x)
        return output
discriminator = Discriminator()
```

__init__(). is used to construct the model. You must first run super (). Run. init__() from nn with __init__(). Module. The discriminator is an MLP neural network that has been sequentially defined using nn. Sequential(). It features the following things:

The input is two-dimensional, and the 256 neurons in the first hidden layer are activated by ReLU.

There are 128 and 64 ReLU-activated neurons in the second and third hidden layers, respectively. A single neuron with sigmoidal activation serves as the output to represent a probability. To prevent overfitting, apply dropout after the first, second, and third hidden layers.

Finally, you use.forward() to explain how the model's output is computed. Here, x stands for the model's input, a tensor in two dimensions. In this method, the output is produced by directly feeding the input x into the given model.

You must create a Discriminator object after defining the discriminator class:

Implementing the Generator:

The generator in generative adversarial networks is the model that uses samples from a latent space as input and produces data that resembles the training set. This model has a two-dimensional input that will in this case receive random points (z1, z2) and a two-dimensional output that will need to produce points (x1, x2) that resemble those from the training set.

Similar to how we implemented the discriminator, so is the implementation of generator . In order to define the neural network architecture, we constructed a Generator class that derives from nn. Module, then we created a Generator object:

In [11]:

```
class Generator(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(8, 16),
            nn.ReLU(),
            nn.Linear(16, 32),
            nn.ReLU(),
            nn.Linear(32, 8),
        )

    def forward(self, x):
        output = self.model(x)
        return output

generator = Generator()
```

Generator in this context refers to the generator neural network. It consists of a linear activation layer with two neurons in the output, two hidden layers with 16 and 32 neurons each, and ReLU activation on all of them. By doing this, the result will be a vector with two elements that can each have a value between -infinity and infinity, denoting (x1, x2).

Training the Models

You configure the following parameters here:

The learning rate (lr), which is set in line 1, is which will be used to modify the network weights.

Line 2 specifies the number of epochs (num epochs), which determines how many training repetitions will be made utilizing the entire training set.

The binary cross-entropy function BCELoss(), which is the loss function used to train the models, is given the variable loss function at line 3.

```
lr = 0.001
num_epochs = 300
loss_function = nn.BCELoss()
```

Because it takes into account a binary classification problem, the binary cross-entropy function is an appropriate loss function for training the discriminator. Given that the generator receives its output from the discriminator, which produces a binary observable output, it is also appropriate for training the generator.

In torch.optim, PyTorch implements multiple weight update strategies for model training. To train the generator and discriminator models, utilize the Adam algorithm. Run the following commands to generate the optimizers using torch.optim:

```
In [13]: optimizer_discriminator = torch.optim.Adam(discriminator.parameters(), lr=lr)
         optimizer_generator = torch.optim.Adam(generator.parameters(), lr=lr)
```

In order to minimize the loss function, we designed a training loop where training samples are provided to the models and their weights are updated

At each training iteration for GANs, the discriminator and generator parameters are updated. The training procedure is divided into two loops, one for the training epochs and the other for the batches for each epoch, as is customary for all neural networks. We started gathering the data to train the discriminator inside the inner loop:

Line 2: From the data loader, we obtained the real samples for the current batch, which you then set to real samples. Keep in mind that batch size is mirrored in the first dimension of the tensor. Each line of the tensor represents a sample from the batch, which is how data are often organized in PyTorch.

Line 4: Labels with the value 1 are created for the real samples using torch.ones(), and they are then assigned to real samples labels.

The generated samples are produced by storing random data in latent space samples, which are then fed into the generator to produce generated samples, as seen in lines 5 and 6.

Line 7: The labels for the created samples are given the value 0 using torch.zeros(), and the labels are then stored in generated samples labels.

```
def train(train_loader,num_epoches):
  for epoch in range(num_epochs):
    for n, (real_samples, _) in enumerate(train_loader):
      # Data for training the discriminator
      real_samples_labels = torch.ones((batch_size, 1))
      latent_space_samples = torch.randint(0,205,(batch_size, 8)).float()
      generated_samples = generator(latent_space_samples)
      generated_samples_labels = torch.zeros((batch_size, 1))
      all_samples = torch.cat((real_samples, generated_samples))
      all_samples_labels = torch.cat(
        (real_samples_labels, generated_samples_labels)
      )

      # Training the discriminator
      discriminator.zero_grad()
      output_discriminator = discriminator(all_samples)
      loss_discriminator = loss_function(output_discriminator, all_samples_labels)
      loss_discriminator.backward()
      optimizer_discriminator.step()

      # Data for training the generator
      latent_space_samples = torch.randint(0,205,(batch_size, 8)).float()

      # Training the generator
      generator.zero_grad()
      generated_samples = generator(latent_space_samples)
      output_discriminator_generated = discriminator(generated_samples)
      loss_generator = loss_function(output_discriminator_generated, real_samples_labels)
      loss_generator.backward()
      optimizer_generator.step()

      # Show loss
      if epoch % 10 == 0 and n == batch_size - 1:
        print(f"Epoch: {epoch} Loss D.: {loss_discriminator}")
        print(f"Epoch: {epoch} Loss G.: {loss_generator}")
```

The real and generated samples and labels are concatenated and stored in the all samples and all samples labels variables, which will be used to train the discriminator, in lines 8 to 11.

Next, you train the discriminator on lines 14 through 19.

Line 14: To prevent gradients from building up in PyTorch, they must be cleared at every training step. To accomplish this, use.zero grad ().

Line 15: The training data in all samples are used to calculate the discriminator's output.

Lines 16 and 17: The labels in all samples labels and the output from the model in output discriminator are used to calculate the loss function.

Line 18: Here we have used loss discriminator.backward to calculate the gradients to update the weights ().

Then Call the optimizer discriminator step to update the discriminator weights ().

The data for the generator's training is, therefore, ready at line 22. With a number of lines equal to the batch size, you store random data in latent space samples. Given that the generator needs two-dimensional data as input, you utilize two columns.

In lines 25 to 32, you train the generator:

Line 25:zero grad is used to clear gradients ().

Line 26: Latent space samples are used to feed the generator, and generated samples is where you keep the output.

Line 27: The output of the discriminator is fed into the output of the generator, which is then stored in output discriminator generated, which serves as the output of the entire model.

Lines 28 to 30: The real samples label variable, which contains labels that are equal to 1, and the output of the classification system stored in the output discriminator generated are used to construct the loss function.

```
[58]
length=0
temp=len(data)
while length<=20000:
  train_loader = torch.utils.data.DataLoader(
    data[length:length+4000], batch_size=batch_size, shuffle=True)
  train(train_loader,num_epochs)
  print(length)
  length=length+4000
  #time.sleep(3)
```

We are using the above code to train the GAN using a dataset with tensors and formatting it to train loader for training the model.

Experimental Results:

Our GAN is learning passwords of rockyou data set and guessing them and predicting new passwords using the rockyou passwords and generating new passwords that are immune to hacking models which are trained with rockyou dataset.

```
1 for val in data[0:1000]:
2   prediction = generator(val[0].float())
3   print(get_predicted_string(prediction))
4
```

```
¶¾¶K4Ð¶¶
aÙP%-pº¶
@  Ñm  î)}
÷Ø(¢¶«l%
 §J`ìÒ±¶
¶®ý¹¥;Y!
z  å.1  &`
  ýRöÄ/
¶dê¶¶¶¶
.J  t¿6ý¾
,Qù  írÝo
  ¶¶¶£Ho½
n-R4¶e<¶
æ  ã¿ ¿
  ¶,t%àÉª
|;£YJg¶¾
Qh£  ;k×À
¶¶JÎJ¶8"
ü$®´/  ¤3
³qw¤§Y>
```