Generating Privacy Preserving Synthetic Datasets

Author: **Nicholas André G. Johnson**

Submitted: **May 13, 2019**

Advisor: **Professor Prateek Mittal**

Second Reader: **Professor Jianqing Fan**

Submitted in partial fulfillment

of the requirements for the degree of

Bachelor of Science in Engineering

Department of Operations Research

and Financial Engineering

Princeton University

I hereby declare that this Independent Work report represents my own work in accordance with University regulations.

Nicholas André G. Johnson

Nicholas Johnson

Generating Privacy Preserving Synthetic Datasets

Nicholas André G. Johnson

Abstract

Given some dataset containing potentially private user information, non-interactive private data release refers to the publishing of a perturbed dataset that preserves the privacy of individual users who have contributed to the true dataset. Privacy is quantified using (ε, δ) differential privacy. This framework raises a natural question: for a fixed privacy tolerance, how can the released dataset be constructed to maximize downstream utility for analytic tasks? Generative models have previously been used to address this problem. However many of these models, for instance RON-Gauss [1], make rather strong assumptions on the nature of the underlying distribution that the data is drawn from. In this work, I make use of Generative Adversarial Networks to develop generative models to solve this problem without making assumptions on the underlying distribution. I draw on composition under Renyi Differential Privacy to establish strong privacy guarantees. I refer to this method as DP-GAN. It is demonstrated empirically that for practical privacy budgets, DP-GAN produces synthetic datasets with greater utility than those produced by RON-Gauss. Furthermore, the performance of a hybrid method that combines DP-GAN and RON-Gauss, named RON-DP-GAN, is empirically investigated and suggests avenues for further development of methods to tackle this problem.

Acknowledgements

Thank you to my advisor Professor Prateek Mittal for his guidance and his steadfast, unwavering support throughout this project. Thank you to Sameer Wagh and Liwei Song for their willingness to meet with me to reflect on the results of my work and to help in tackling challenges. Thank you to Professor Ryan Adams for his willingness to engage with me regarding Generative Adversarial Networks and to Professor Jianqing Fan for his support in my pursuit of this project. Finally, thank you to Jean Bausmith for her management of Electrical Engineering Independent Work and support during this journey.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A defining feature of the Big Data era in which we presently live is the collection of enormous quantities of user information by countless companies and organizations that engage in some form with humans. Whether we consider large tech companies, healthcare providers, financial institutions or other companies from almost any domain, the large scale collection and aggregation of user data is a common trend. From the perspective of a company, this practice is very sensible; collecting user information to create a large dataset allows a company to make use of large scale analytics to better understand the user experience and ultimately develop a better product for the user. For instance, Google collects Chrome usage data from its users in order to refine the features available through their internet browser. However, this practice naturally gives rise to considerations related to data privacy. Specifically, although a company can make use of the macroscale characteristics of a collected dataset to improve their product, said company could quite possibly be collecting sensitive personal information from its users during its data collection process. Any given user might not want this personal information to be accessible by an employee of the company, let alone to the larger public. Beyond the inherent ethical concerns in this situation, a company is incentivized to offer certain data privacy guarantees to their users as doing so would make the users more likely to consent to having their data aggregated into the dataset, thereby increasing the company's ability to improve their product. Thus, this leads to the following problem: how to preserve the macroscale statistical properties of a dataset while limiting the amount of information an analyst can infer about any single user who's data is contained in the dataset?

To tackle this problem, Cynthia Dwork proposed the notion of Differential Privacy (DP) to rigorously quantify the extent to which a mechanism operating on a dataset preserves the privacy of individual contributing users [2]. DP has been widely adopted by both academics and industry professionals to quantify data privacy, notably being used by Google and Apple. The formal definition DP will be presented in the following section; for now, it suffices to think of a differentially private mechanism as a mechanism that adds a certain amount of random noise to statistical queries made by analysts. Within the framework of DP, there are two settings: the interactive setting and the non-interactive setting. In the interactive setting of DP, an analyst sequentially makes queries to a dataset and after each query, the differentially private mechanism returns a slightly randomly perturbed version of the true query value. The primary limitation of this setting is that the number of queries made by the analyst must be finite and known a priori when designing the mechanism to obtain meaningful DP guarantees. The alternate non-interactive setting, which will be the focus of this paper, exploits the immunity of DP to post-processing. This immunity means that given a query with certain DP guarantees, all further operations performed on that query will have the same DP guarantees with respect to the initial dataset [3]. The non-interactive setting can be thought of as follows: firstly, an analyst in the interactive setting requests to see the entire dataset as a query and secondly, the resulting dataset returned by the differentially private mechanism is made publicly available. Analysts are free to conduct an unlimited number of queries on the released privatized dataset and all outputs of such queries will have DP guarantees. This non-interactive setting is more desirable than its interactive counterpart because no assumptions are made on the number or the nature of queries made by analysts. However, as a result of this lack of assumptions, significant perturbations must typically be made to the dataset in order for the released version to have strong DP guarantees. This poses a challenge as there is a tradeoff between the privacy of the users who contributed to the dataset and the utility of the released data: adding greater noise to the data enhances privacy but diminishes utility.

Significant public outcry resulting from previous failures of non-interactive private data release suggest the need for more robust and sophisticated methods. In 2006, Netflix announced a contest in which they released a supposedly anonymized dataset containing usage data of roughly 500,000 users and challenged teams to develop an algorithm that could outperform their in house movie recommendation algorithm by 10% [4]. A few weeks after this announcement, a team of researchers from the University of Texas at Austin demonstrated that this dataset could in fact be de-anonymized and personal information from specific users could be identified [5]. Beyond being a clear failure on the part of Netflix to respect its users' privacy, this release was arguably in violation of the 1988 Video Privacy Protection Act. Although this competition proceeded in 2006, Netflix cancelled its second iteration of this contest in 2010 following a lawsuit filed by an in-the-closet lesbian mother citing privacy violations [4]. The Netflix fiasco clearly demonstrates the importance of non-interactive private data release. Although the data in question was video viewing data which some might deem to not be overly sensitive in nature, one could easily imagine a similar circumstance occurring with health or demographic data. To this end, this work builds off existing work in this space and draws heavily on generative modelling to develop utility enhancing privacy preserving mechanisms for non-interactive private data release.

Chapter 2

Background

2.1 Differential Privacy

2.1.1 (ε, δ)-Differential Privacy

The purpose of this section is to formally present Differential Privacy (DP) and to comment on some of its properties and variants. Before defining DP, the notion of adjacent datasets must be established. I will use the term record to refer to the entirety of a single user's data and the term dataset to refer to a collection of multiple records. Let $\mathbf{D}$ denote the set of all possible datasets. Two datasets $d, d' \in \mathbf{D}$ are adjacent if they differ exclusively in the inclusion or deletion of a single record. For fixed $d \in \mathbf{D}$, let $adj(d) \subseteq \mathbf{D}$ be the set of all datasets that are adjacent to d. Let $\mathscr{A} : \mathbf{D} \to \mathbf{A}$ be a randomized algorithm that maps datasets to some arbitrary output space (denoted here by $\mathbf{A}$). The algorithm $\mathscr{A}$ is said to be (ε, δ)-differentially private, or(ε, δ)-DP, if $\forall d \in \mathbf{D}, \forall d' \in adj(d)$ and $\forall S \subseteq \mathbf{A}$, we have:

$$\Pr[\mathscr{A}(d) \in S] \leq e^{\varepsilon} \Pr[\mathscr{A}(d') \in S] + \delta$$

where the probabilities are taken over everything that is random in the algorithm $\mathscr{A}$ [3]. This definition means that the probability of any given outcome when $\mathscr{A}$ acts on a dataset $d \in \mathbf{D}$ is very close to the probability of the exact same outcome occurring when $\mathscr{A}$ acts on a neighboring dataset $d' \in \mathbf{D}$, and the parameters ε and δ quantify exactly how close these two probabilities are in the worst case. DP was originally formulated without the parameter δ which was only later introduced to allow for more effective composition of multiple differentially private algorithms. Typically, ε is referred to as the privacy budget while δ is often referred to as the probability of failure. Intuitively, ε and δ quantify the extent to which the output of $\mathscr{A}$ can depend on any single record in a dataset. When $\varepsilon = 0$ and $\delta = 0$, the two probabilities are exactly equal and privacy is preserved perfectly, however the algorithm cannot infer any additional information from the inclusion of an additional record in a dataset. Smaller values of ε and δ correspond to more stringent privacy guarantees. If we define the privacy loss to be the random variable $Z = \ln\left(\frac{\Pr[\mathscr{A}(d) \in S]}{Pr[\mathscr{A}(d') \in S]}\right)$, then if $\mathscr{A}$ is (ε, δ)-differentially private we have $\Pr[Z > \varepsilon] \leq \delta$ where the probability is taken over all that is random in $\mathscr{A}$, a random choice of $d \in \mathbf{D}$ and a random choice of $d' \in adj(d)$. A useful property of DP is post-processing which was referenced in the previous section. Formally, post-processing is defined as follows: given an (ε, δ)-DP algortihm $\mathscr{A}$ with range $\mathbf{A}$, $\forall f : \mathbf{A} \rightarrow range(f), f(\mathscr{A}(\cdot))$ is (ε, δ)-DP [3].

2.1.2 Composition Theorems

Composition properties are needed in DP in order to formalize the DP guarantees that result from algorithms that apply multiple DP algorithms sequentially. Here we will formally present the basic composition theorem, the advanced composition theorem and briefly reference the moments accountant method. Basic composition is rather straightforward and intuitive as its name implies: suppose algorithm $\mathscr{A}_1$ is $(\varepsilon_1, \delta_1)$-DP and algorithm $\mathscr{A}_2$ is $(\varepsilon_2, \delta_2)$-DP, then the composition of these two algorithms is $(\varepsilon_1 + \varepsilon_2, \delta_1 + \delta_2)$-DP [3]. It is easy to imagine how basic composition can be applied to composing N differentially private algorithms.

Although basic composition is intuitive, it is inconvenient because for the composition of a relatively small number of differentially private algorithms, the privacy budget grows very quickly (specifically, $\varepsilon = \mathscr{O}(N)$ when N algorithms are being composed). Consequently, the advanced composition theorem was developed to more tightly bound the growth of the privacy budget under N-fold composition when the probability of failure of the individual algorithm is 0. Under advanced composition, the composition of N algorithms that are $(\varepsilon, 0)$-DP is (ε', δ)-DP where

$$\varepsilon' = \varepsilon\sqrt{2N\ln(\frac{1}{\delta})}$$

$\forall\delta \geq 0$ [3]. We omit the presentation of the generalized advanced composition theorem for N-fold composition of (ε, δ) algorithms where $\delta \neq 0$.

Although advanced composition achieves a tighter bound under N-fold composition than basic composition (advanced composition gives $\varepsilon = \mathscr{O}(\sqrt{N})$), tighter bounds are often desired particularly for the commonly used Gaussian mechanism [3]. In 2016, Dr. Martin Abadi introduced the Moments Accountant Method for N-fold composition of DP algorithms based on the Gaussian mechanism, motivated by the desire to have tight privacy bounds when composing a large number of (ε, δ)-DP algorithms [6]. This method achieves tighter bounds than the advanced composition theorem and is particularly useful in calculating privacy bounds for neural networks where each iteration of gradient descent is made differentially private. This method will not be formally presented in this paper.

2.1.3 Renyi Differential Privacy

Despite the even tighter privacy bounds that result from the Moments Accountant Method, it is possible to do even better. Doing so however requires the introduction of an alternate definition of DP formulated in 2017, known as Renyi DP [7]. Renyi DP is founded on the notion of Renyi Divergence. Formally, the Renyi Divergence of order $\alpha \geq 1$ of two probability distributions P and Q over the set of real numbers $\mathbf{R}$ is defined as:

$$D_\alpha(P||Q) := \frac{1}{\alpha - 1}\ln\left(\mathbb{E}_Q\left[\frac{P(x)}{Q(x)}\right]^\alpha\right)$$

The Renyi divergence of order $\alpha = 1$ is defined by continuity and is equal to the Kullback-Leibler Divergence:

$$D_1(P||Q) = \mathbb{E}_P\left[\ln\left(\frac{P(x)}{Q(x)}\right)\right]$$

Having defined Renyi Divergence, we can now formally define Renyi DP. A randomized algortihm $\mathscr{A} : \mathbf{D} \rightarrow \mathbf{R}$ is said to be (α, ε)-RDP (Renyi Differentially Private) if $\forall d \in \mathbf{D}$ and $\forall d' \in adj(d)$, we have:

$$D_{\alpha}(\mathscr{A}(d)||\mathscr{A}(d')) \leq \varepsilon$$

Renyi DP is monotonic in α: for $\alpha_1 \geq \alpha_2$, we have (α_1, ε)-RDP $\implies$ (α_2, ε)-RDP. Renyi DP also satisfies the following nice composition theorem: suppose algorithm $\mathscr{A}_1$ is (α, ε_1)-RDP and algorithm $\mathscr{A}_2$ is (α, ε_2)-RDP, then the composition of these two algorithms is $(\alpha, \varepsilon_1 + \varepsilon_2)$-RDP [7].

At this point the reader may be wondering why Renyi DP is helpful. Renyi DP is related to the more familiar (ε, δ)-DP by the following property: suppose $\mathscr{A} : \mathbf{D} \rightarrow \mathbf{R}$ satisfies (α, ε)-RDP, then $\mathscr{A}$ satisfies $\left(\varepsilon + \frac{\ln(\frac{1}{\delta})}{\alpha - 1}, \delta\right)$-DP $\forall \delta \in (0,1)$. Thus, given any algorithm satisfying (α, ε)-RDP, we can obtain an (ε, δ)-DP guarantee by choosing a desired value δ and then deriving the corresponding ε from this property. Composing private mecahnisms in (α, ε)-RDP form and then converting them to (ε, δ)-DP form results in tighter privacy bounds than those achieved by the Moments Accountant Method for large numbers of compositions. Further details can be found in Ilya Mironov's original publication [7].

2.2 RON-Gauss

RON-Gauss is a method for non-interactive private data release developed by Thee Chanyaswad in 2017 that leverages the Diaconis-Freedman-Meckes (DFM) effect and the gaussian generative model [1]. The DFM effect states that random orthonormal projections of high-dimensional data, under certain mild conditions, are nearly gaussian. At a high level, given some dataset $d \in \mathbb{R}^{nxm}$, RON-Gauss first centers it by calculating and subtracting its mean $\mu_{DP} \in \mathbb{R}^{nx1}$ using the common Laplacian mechanism to guarantee DP [3]. The centered dataset is then projected onto a random subspace defined by $U \in \mathbb{R}^{nxn'}$ where U has orthonormal columns and $n' << n$. The covariance matrix $\Sigma_{DP} \in \mathbb{R}^{n'xn'}$ of the projected data is calculated again using the Laplacian mechanism to guarantee DP. Having done this, a synthetic dataset of size m can be created by drawing m samples from the distribution $N(0, \Sigma_{DP})$, reconstituting the samples to the original input space and finally adding back the mean μ_{DP}. RON-Gauss guarantees $(\varepsilon, 0)$-DP [1].

The primary strength of this approach is that by combining the DFM effect with the gaussian model, the algorithm obtains a reasonable approximation of the dataset by only estimating two parameters. This avoids problems related to rapid privacy budget increases in response to the composition of large numbers of differentially private computations. Despite the guarantees of the DFM effect, this model suffers from the fact that the random projection of the initial dataset is a lossy transformation and because even after projection, the resulting data set may not in fact be gaussian.

2.3 Generative Adversarial Networks

This work draws heavily on Generative Adversarial Networks (GANs), a method for estimating generative models that was first proposed by Ian Goodfellow in 2014 and has been widely accepted in the machine learning community [8]. Suppose we are given a dataset $\{x_i\}_{i=1}^N$ and assume that each $x_i \in \mathbb{R}^m$ is drawn from some underlying distribution $p_{data}(x)$. Let $p_Z(z)$ denote some arbitrary prior distribution over vectors $z \in \mathbb{R}^q$ where $q << m$. Let $G : \mathbb{R}^q \to \mathbb{R}^m$ and $D : \mathbb{R}^m \to [0, 1]$. The function G is referred to as the generator and the function D is referred to as the discriminator. In the GAN formulation, the generator and the discriminator compete in a two player game. The goal of the generator is to generate samples that mirror those drawn from the underlying probability distribution when given samples from the prior distribution, while the goal of the discriminator is to correctly determine which samples come from the true underlying distribution and which samples are generated by the generator. Formally, the generator and the discriminator play the following minimax game:

$$\min_G \max_D \mathbb{E}_{x \sim p_{data}(x)}[\ln(D(x))] + \mathbb{E}_{z \sim p_Z(z)}[\ln(1 - D(G(z)))]$$

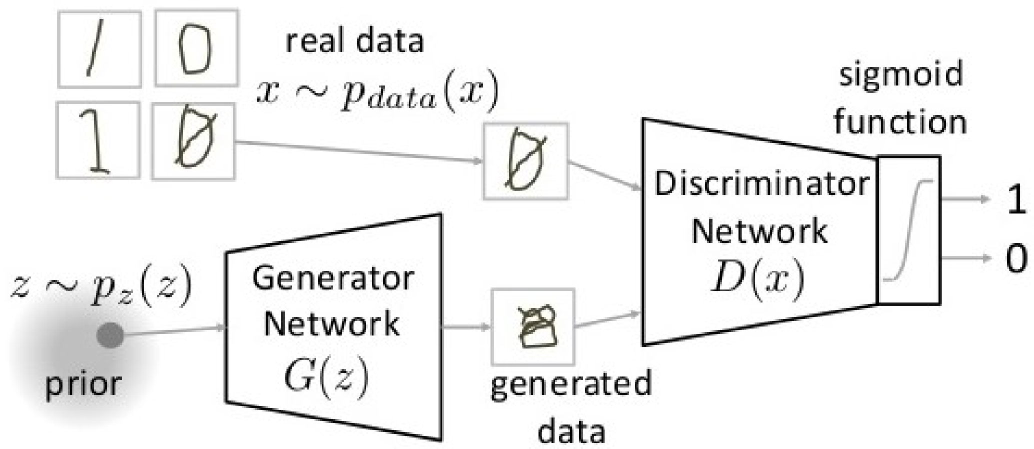


Fig. 2.1. GAN Schematic

In his seminal paper, Goodfellow demonstrated that if the generator and discriminator are allowed to be arbitrary functions, there exists a single Nash Equilibrium for the game where we have: $G(Z) \sim p_{data}(x)$ and $D(G(Z)) = D(X) = \frac{1}{2}$ [8]. Thus, at equilibrium the distribution defined by generator function and the prior distribution perfectly mirrors the underlying distribution and the discriminator is unable to distinguish between "real" and "fake" examples. This result is preserved if we constrain the generator and discriminator to be neural networks due to the universal approximation theorem [9].

After constraining the generator and the discriminator to be neural networks, common practice is to define $p_Z \sim N(0,1)$ and then alternate between training the generator and training the discriminator iteratively using an optimizer. After training, samples can be drawn from the generator that mirror those drawn from the underlying distribution. GANs are thus an attractive tool to aid in non-interactive private data release because of their ability to model any given distribution without making assumptions on said distribution as was done by RON-Gauss for instance. Figure 2.1 is a schematic of the GAN setup [10].

Unfortunately, although the theoretical result proved by Goodfellow suggests that the generator should ultimately be able to mirror any underlying probability distribution, training the generator and the discriminator in practice results in several additional challenges. Most notably, training GANs can sometimes be unstable if one network learns faster than the other one. The discriminator network often becomes stronger much more rapidly than the generator and can impede generator learning in the early stages of training if the discriminator is too strong [11].

In 2016, Alec Radford empirically demonstrated interesting behaviour of trained generators in response to deliberate manipulation of the input vector used to sample from the generator [12]. Radford took a generator G trained to generate images of human faces and identified sets of vectors (z_1, z_2, z_3) such that $G(z_1)$ was an image of a man with glasses on, $G(z_2)$ was an image of a man without glasses on and $G(z_3)$ was an image of a woman without glasses on. For almost all such sets, Radford demonstrated that $G(z_1 - z_2 + z_3)$ was an image of a woman wearing glasses. This behaviour suggested that the prior distribution plays a significant role in the output of the generator, an observation that motivated one of the main contributions of this work.

2.4 Neural Networks and Differential Privacy

The purpose of this section is to formalize what it means for a neural network to be (ε, δ)-DP. Let the function $f(\cdot|\theta) : \mathscr{X} \to \mathscr{Y}$ be a neural network where $\theta \in \mathbb{R}^p$ denotes the learned parameters of the network. Suppose we have a training set $\{(x_j, y_j)\}_{j=1}^n$ where $x_j \in \mathbb{R}^m$ and we use an arbitrary optimization procedure $\mathscr{O}$ to train the neural network (examples of $\mathscr{O}$ are stochastic gradient descent, Adam and adagrad). To cast this in a DP framework using the notation from the earlier setting, let $d = [x_j(i)]_{ij} \in \mathbb{R}^{mxn}$, $\mathbf{D} = \bigcup_{k=1}^{\infty} A_k$ where $A_k = \{a \in \mathbb{R}^{mxk}\}$ and interpret $\mathscr{O}$ as a randomized algorithm that maps $\mathbf{D} \to \mathbb{R}^p$. Note that $\mathscr{O}$ also requires the training labels y_j to output a set of parameters θ^*, however we can ignore this as we are focusing on DP in the context of the training examples x_j exclusively and not the training labels as well. Within this framework, we can say that the neural network $f(\cdot|\theta)$ is (ε, δ)-DP if $\forall d \in \mathbf{D}, \forall d' \in adj(d)$ and $\forall \theta^* \in \mathbb{R}^p$, we have:

$$\Pr[\mathscr{O}(d) = \theta^*] \leq e^{\varepsilon} \Pr[\mathscr{O}(d') = \theta^*] + \delta$$

where the probabilities are taken over everything that is random in $\mathcal{O}$. Intuitively, this limits the extent to which the parameters of the network can be dependent on any single record in the dataset.

Standard optimization procedures like stochastic gradient descent must be altered in order to provide DP guarantees for trained networks. In 2019, a team of researchers from Google released a GitHub repository called Tensorflow Privacy and an accompanying white paper [13]. Their work consisted of creating (ε, δ)-DP implementations of many of the most common neural network optimization procedures using composition under RDP. At a high level, a classical optimization engine can be transformed into a private version by clipping the gradients of the training sample at each training iteration using some pre-defined maximum value (selected as a hyperparameter) and thereafter adding gaussian noise to the gradients before computing the parameter updates. Further details can be found in their paper [13].

Chapter 3

Main Contributions

This work presents three primary contributions:

- A differentially private GAN (DP-GAN) is trained to generate private synthetic data exploiting RDP to generate tight privacy bounds.
- Empirical evidence is presented that suggests DP-GAN is more effective for non-interactive private data release than is RON-Gauss.
- The effectiveness of a hybrid of DP-GAN and RON-Gauss, referred to as RON-DP-GAN, for non-interactive private data release is empirically investigated.

Although the idea of creating differentially private GANs has been explored before, previous attempts have either included flawed privacy guarantees [14] or employed epsilon values that were too generous ($\varepsilon \geq 4.0$) [15] [16]. This work draws on RDP based optimization engines from the Tensorflow Privacy GitHub repository to establish reliable privacy guarantees for $\varepsilon \in [1,3]$ and $\delta = 10^{-5}$. Since after training a GAN, we are only interested in keeping the generator and discarding the discriminator, only the generator needs to be made differentially private with respect to the training data. Doing so can be achieved by using a DP optimizer to train the generator and a regular, non-private optimizer to train the discriminator. However, doing so further compounds the stability challenges inherent to GAN training because introducing noise into the generator gradient updates further weakens the strength of the generator with respect to the discriminator, particularly during the early stages of training. To address this challenge, I introduced gaussian noise to the ouput of each layer of the discriminator (with the exception of the final layer) as a form of regularization. This has the effect of facilitating generator learning in part by preventing the discriminator from becoming too strong too rapidly.

When evaluating utility in the non-interactive private data release setting, common practice is to first select some machine learning task (or a collection of tasks) and perform the task on the true dataset. The same task is subsequently performed on the private dataset and the extent to which the resulting output differs from the output produced by performing the task on the true dataset is a measure of utility [1]. In this work, utility was evaluated when RON-Gauss and DP-GAN were respectively used for generation of the synthetic dataset for $\varepsilon \in [1,3]$ and for all tested values of ε, DP-GAN had better utility than RON-Gauss for the selected machine learning task.

RON-DP-GAN is a hybrid approach inspired by the impact of the prior input vector on GAN behaviour discovered by Radford [12]. In RON-DP-GAN, a DP-GAN is trained with $p_Z \sim N(\mu_{DP}, \Sigma_{DP})$ rather than the usual $p_Z \sim N(0,1)$ where μ_{DP} and Σ_{DP} are the parameters computed by RON-Gauss when applied to the true dataset. Given a fixed privacy budget ε, ε_{RON} and ε_{GAN} are allocated to deriving the differentially private parameters that defines the prior and to the iterative training of the GAN respectively, where $\varepsilon_{RON} + \varepsilon_{GAN} = \varepsilon$.

Chapter 4

Methodology

4.1 (DP-)GAN Architecture

The design of the GAN employed in this work follows the design studied by Alec Radford [12]. The GAN code used in this work was modelled from code in the tensorflow-MNIST-GAN-DCGAN GitHub repository [17]. The generator is composed of a 100 dimensional prior input vector and 5 deconvolutional layers, the first 4 of which employ batch normalization followed by the Leaky ReLu activation function with parameter 0.2. The last layer uses the tanh activation function. The discriminator employs 5 convolutional layers the first 4 of which use the Leaky ReLu activation function with parameter 0.2. Convolutional layers 2, 3 and 4 include batch normalization prior to the application of the activation function. The last layer uses the sigmoid activation function. In addition, the output of each Leaky ReLU activation function in the discriminator is perturbed by adding gaussian noise with mean 0 and variance 0.5. Figure 4.1 is a schematic of the generator (denoted by G) and the discriminator (denoted by D) architecture which includes the dimensions of the filters and output at each layer of each network (note Figure 4.1 omits the gaussian noise layers in the discriminator)[17].

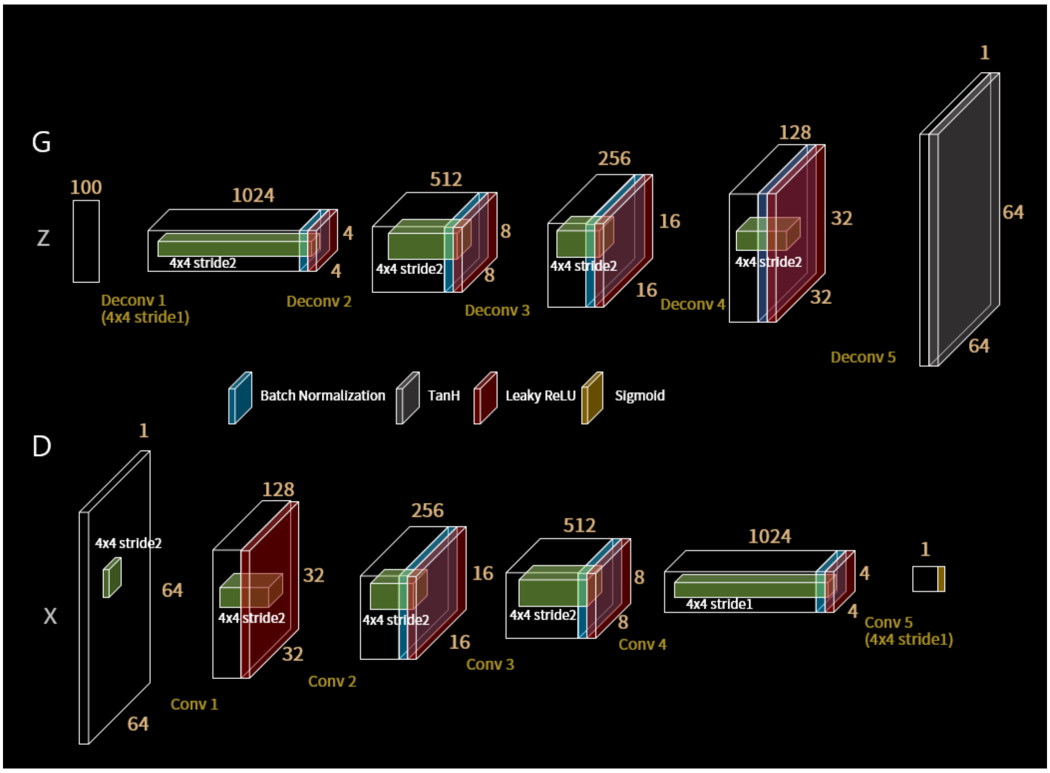


Fig. 4.1. Generator and Discriminator Architecture

4.2 (DP-)GAN Training Procedure

All GANs in this work were trained to generate handwritten digits from the MNIST distribution, using 55000 entries in MNIST as the training set. The training set was normalized and all weights in the network were initialized to have mean 0 and standard deviation 0.02. The batch size was set to 100. For the non private GAN, the Adam optimizer was used to train the discriminator and the generator with learning rate set to 0.0002 and $\beta_1 = 0.5$. For DP-GAN, Adam with the aforementioned parameters was again used to train the discriminator. However, the generator was trained using the DPAdamGaussianOptimizer from the Tensorflow Privacy library [18]. In addition to setting the learning rate to 0.0002 and $\beta_1 = 0.5$, the parameter l2_norm_clip was set to 1.5 and num_microbatches was set to 50. Finally, the value of the parameter noise_multiplier varied depending on the privacy budget of a given GAN and was calculated using the compute_rdp function from Tensorflow Privacy [18].

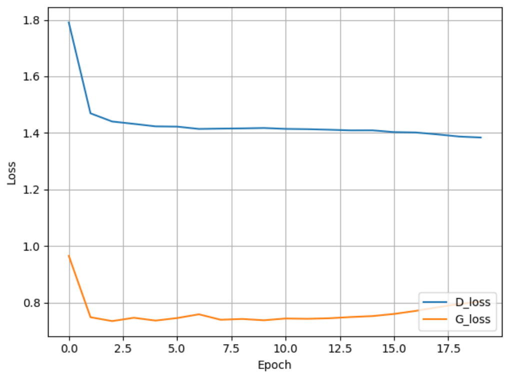
4.3 Evaluating Utility

To evaluate the performance of RON-Gauss and DP-GAN for non-interactive private data release, I posited that an analyst was attempting to determine the first principal component of a true dataset given access to a representative synthetic dataset by calculating the first principal component of the synthetic data. Utility was measured as the Euclidean norm of the difference between the first principal component of the synthetic dataset and the first principal component of the true dataset. Experiments were performed for $\varepsilon \in \{1.0, 1.5, 2.0, 2.5, 3.0\}$. In the case of DP-GAN and RON-DP-GAN, $\delta = 10^{-5}$ for all experiments. For RON-Gauss, we have $\delta = 0$ by definition. Three trials were performed for each setting of ε and δ when evaluating utility for DP-GAN and RON-DP-GAN; fifty trials were performed for each setting of ε and δ when evaluating utility for RON-Gauss. On a given trial for DP-GAN or RON-DP-GAN, ten synthentic datasets were generated after training the model and the reported utility is the average of the utility obtained from each of the ten synthetic datasets.

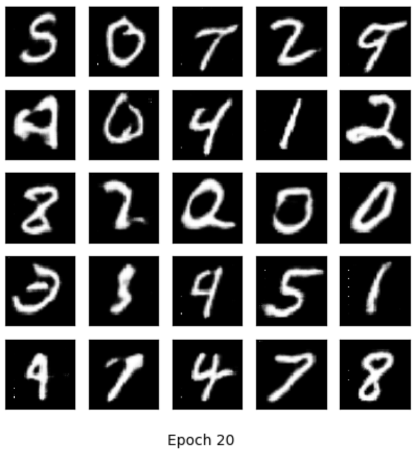
Chapter 5

Results and Discussion

5.1 Non Private GAN



(a) Non Private GAN Training Loss

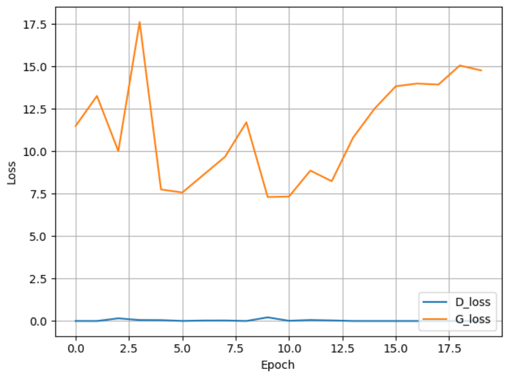


(b) Non Private GAN Samples

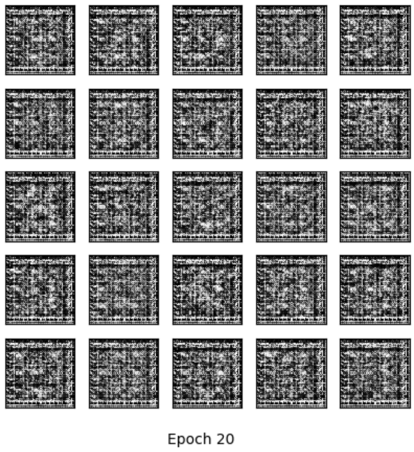
Fig. 5.1. Non Private GAN

Figure 5.1a shows the generator and discriminator loss during training for the Non Private GAN and Figure 5.1b shows samples generated from this model after epoch 20. These illustrations are included primarily to serve as qualitative comparisons to illustrations corresponding to the private version of this model (DP-GAN). When training GANs, it is generally desirable for the generator loss to be less than the discriminator loss and to tend towards zero, which can loosely be observed in the above graph. Note that the generated samples are crisp and look fairly realistic.

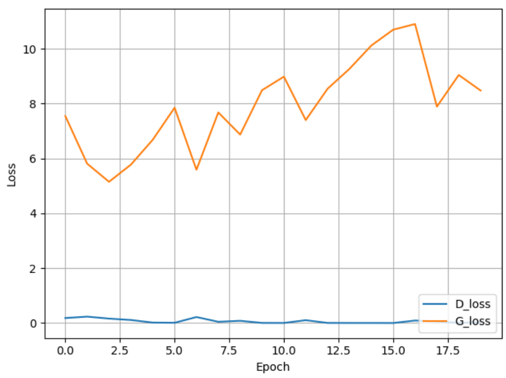
5.2 Training DP-GAN



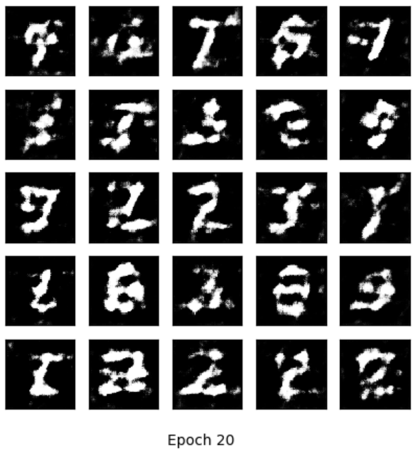
(a) Noiseless DP-GAN Training Loss



(b) Noiseless DP-GAN Samples



(c) Noisy DP-GAN Training Loss



(d) Noisy DP-GAN Samples

Fig. 5.2. DP-GAN

Figures 5.2a, 5.2b, 5.2c and 5.2d show the training loss and generated samples from a DP-GAN that does not have gaussian noise layers in its discriminator and one that does respectively ($\varepsilon = 2.5$ in both cases). We refer to a DP-GAN that does not have gaussian noise layers in its discriminator as a noiseless DP-GAN and refer to a DP-GAN with such layers as a noisy DP-GAN. Although in both cases, the discriminator loss is very close to 0 throughout the training process, the generator loss is lower for the noisy DP-GAN which suggests that the generator is performing better than in the noiseless case. This is expected as adding gaussian noise layers in the discriminator should allow the generator to learn more easily, particularly during the early stages of training, by acting as a regularizer for the discriminator. The effectiveness of adding gaussian noise layers is further supported by the samples that are generated from each model following training. The samples from the noiseless model appear indistinguishable from random noise whereas samples from the noisy DP-GAN look significantly more plausible. Note that the generator loss for the noisy DP-GAN is significantly greater than that for the non private GAN and the images generated by the noisy DP-GAN are qualitatively poorer than those generated by the non private GAN. This is to be expected due to the noisy gradients that the generator receives to ensure privacy guarantees. All further GAN based models referenced in this work include gaussian noise layers in the discriminator.

5.3 Superiority of DP-GAN over RON-Gauss

DP-GAN					
Value of ε:	1.0	1.5	2.0	2.5	3.0
Trial 1	0.501	0.689	0.902	0.783	0.599
Trial 2	0.729	0.836	0.807	0.933	0.618
Trial 3	0.548	0.463	0.697	0.776	0.706
Mean	0.593	0.663	0.802	0.831	0.641
Standard Deviation	0.098	0.154	0.084	0.072	0.047

Table 5.1: DP-GAN Experimental Results

RON-Gauss					
Value of ε:	1.0	1.5	2.0	2.5	3.0
Mean	1.367	1.427	1.340	1.370	1.347
Standard Deviation	0.180	0.218	0.207	0.226	0.223

Table 5.2: RON-Gauss Experimental Results

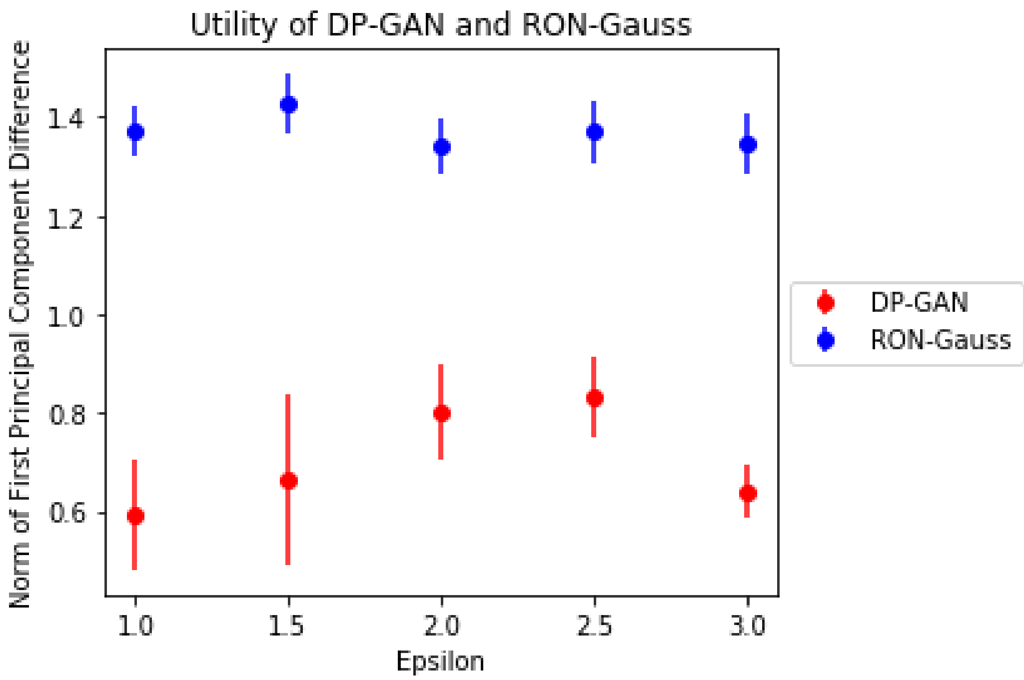


Fig. 5.3. Performance of RON-Gauss and DP-GAN

Tables 5.1 and 5.2, and Figure 5.3 summarize the performance of RON-Gauss and DP-GAN. Notice that for all tested values of ε, DP-GAN produced a synthetic data set with utility that is statistically significantly better than the utility of the synthetic dataset produced by RON-Gauss for the chosen evaluation metric. This suggests that DP-GAN is a superior method for non-interactive private data release than RON-Gauss.

One would expect, for a fixed synthetic dataset generation method, that the utility of the generated dataset would increase as the value of ε increases; less stringent privacy guarantees can be achieved with less significant perturbations to the true dataset. This would imply that, for a fixed method, as the value of ε increases, the Euclidean norm of the difference between the first principal component of the synthetic dataset and that of the true dataset should decrease. However, this trend is not observed in the experimental results of RON-Gauss nor in those of DP-GAN. We postulate two possible explanations for the absence of this trend:

1. A large number of observations must be made to observe this trend due to significant randomness in the synthetic data generation process.

2. The chosen utility measure in this work is not an ideal measure of the true utility of the generated synthetic datasets.

The first possible explanation might explain the absence of the expected trend in the performance of DP-GAN as only three trials were performed for each fixed value of ε. Only three observations were made because each observation requires training a GAN using a DP optimizer which is very computationally expensive - a single trial takes roughly 10 to 12 hours to complete on a NVIDIA Tesla P100 GPU. However, fifty trials were performed for each fixed ε when evaluating RON-Gauss (as trials for RON-Gauss are significantly less computationally expensive than trials for DP-GAN) yet the trend still was not observed. Thus, this first possible explanation seems unlikely. The second possible explanation is more convincing. Utility of the released dataset in non-interactive private data release refers to how well analytic tasks performed on the released dataset approximate the results of the same tasks being performed on the true dataset. As such, utility is extremely dependent on the downstream analytic tasks that will be performed and consequently has no universally accepted metric. This work focused on the performance of a single unsupervised machine learning task to quantify utility. A more robust measure of utility could likely be derived from aggregating the difference in performance on multiple analytic tasks between the true dataset and the synthetic dataset.

5.4 Investigation of RON-DP-GAN

Let $\psi := \dfrac{\varepsilon_{RON}}{\varepsilon} \in (0,1)$ denote the fraction of the privacy budget ε that is allocated to computing μ_{DP} and Σ_{DP}. Experiments for RON-DP-GAN were first performed

RON-DP-GAN ($\psi = 0.4$)					
Value of ε:	1.0	1.5	2.0	2.5	3.0
Trial 1	1.823	1.844	1.875	1.859	1.834
Trial 2	1.823	1.702	1.791	1.847	1.198
Trial 3	1.862	1.413	1.808	1.556	1.396
Mean	1.836	1.653	1.825	1.754	1.476
Standard Deviation	0.018	0.179	0.036	0.140	0.262

Table 5.3: RON-DP-GAN Experimental Results ($\psi = 0.4$)

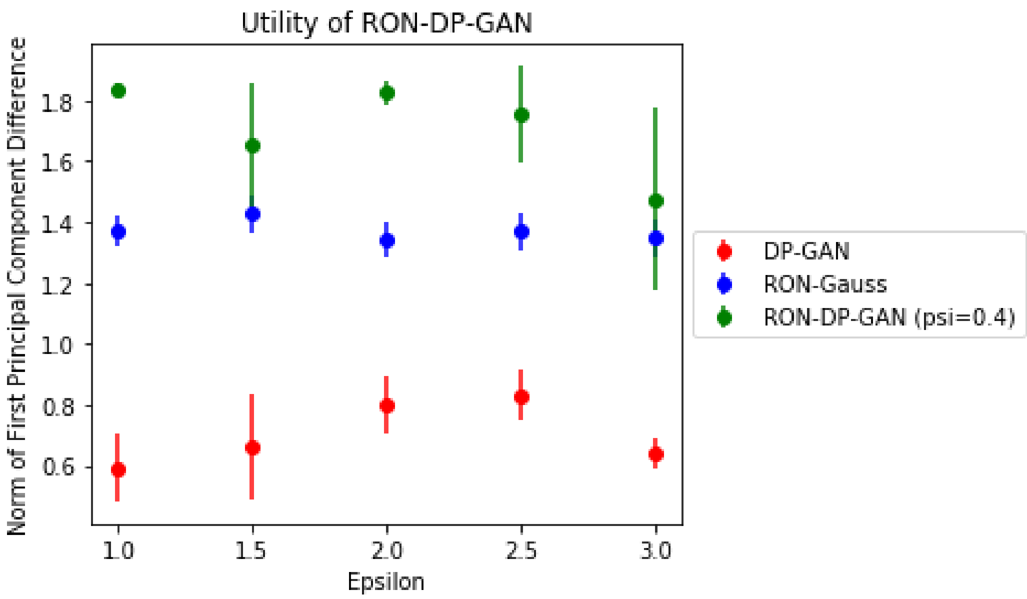


Fig. 5.4. Performance of RON-Gauss, DP-GAN and RON-DP-GAN for $\psi = 0.4$

for $\psi = 0.4$. Table 5.3 and Figure 5.4 summarize the performance of RON-DP-GAN for this value of ψ. RON-DP-GAN performed statistically significantly worse than RON-Gauss for almost all tested values of ε. This result was unexpected. Intuitively, the result suggests that the information RON-DP-GAN gains from using RON-Gauss as a prior during training and sampling is less than the information that is lost due to the decreased privacy budget allocated to training the generator when $\psi = 0.4$. If this intuition is correct, then decreasing the value of ψ should improve the performance of RON-DP-GAN.

RON-DP-GAN ($\psi = 0.15$)					
Value of ε:	1.0	1.5	2.0	2.5	3.0
Trial 1	1.775	0.919	0.829	1.803	1.805
Trial 2	1.817	1.522	1.632	0.956	1.895
Trial 3	1.438	0.749	1.763	1.418	1.291
Mean	1.677	1.063	1.408	1.392	1.664
Standard Deviation	0.170	0.332	0.413	0.346	0.266

Table 5.4: RON-DP-GAN Experimental Results ($\psi = 0.15$)

Table 5.4 and Figure 5.5 summarize the performance of RON-DP-GAN for $\psi = 0.15$. The performance of RON-DP-GAN was again worse than that of DP-GAN for $\psi = 0.15$. However, for almost all tested values of ε, RON-DP-GAN performed better for $\psi = 0.15$ than for $\psi = 0.4$, although in some cases the difference is not statistically significant. This suggests that further lowering of the hyperparameter ψ may improve the performance of RON-DP-GAN, possibly to a point at which RON-DP-GAN would outperform DP-GAN. Note that the experimental results for RON-DP-GAN do not display a monotonic increase in utility (decrease in the measured norm) as ε increases for either $\psi = 0.4$ or $\psi = 0.15$. The postulates put forth in section 5.3 to explain this observation in the case of RON-Gauss and DP-GAN remain valid for RON-DP-GAN.

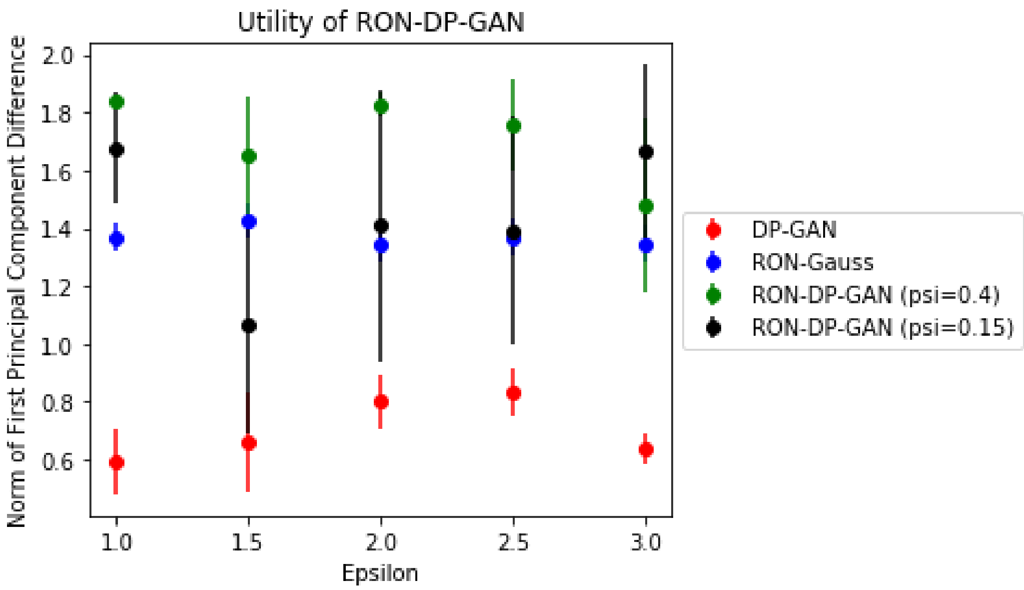


Fig. 5.5. Performance of RON-Gauss, DP-GAN, RON-DP-GAN for $\psi = 0.4$ and RON-DP-GAN for $\psi = 0.15$

Chapter 6

Conclusion

6.1 Main Findings

This work demonstrates that a differentially private GAN (DP-GAN) with tight privacy bounds can be trained to generate synthetic private data by leveraging Renyi Differential Privacy and introducing gaussian noise layers into the GAN's discriminator network. Moreover, it is demonstrated empirically that DP-GAN produces private synthetic datasets that have greater utility than private datasets produced by RON-Gauss. This suggests that DP-GAN is a more effective method for non-interactive private data release. Finally, empirical results indicate that RON-DP-GAN, a hybrid approach of DP-GAN and RON-Gauss, performs worse than DP-GAN alone for $\psi \in \{0.15, 0.4\}$.

6.2 Future Directions

There are several natural future directions for this work. Some of the most promising expansions include:

- Performing further hyperparameter tuning for DP-GAN and RON-DP-GAN.
- Exploring alternate measures of utility.
- Exploring the effects of alternate priors on the performance of DP-GAN.

As discussed in section 5.4, tuning the hyperparameter ψ to a lower value could very likely improve the performance of RON-DP-GAN. Additionally, the value of the maximum allowable gradient norm during gradient updates of the discriminator in DP-GAN and RON-DP-GAN was not tuned (it was taken to be its default value from the optimizer implementation). The influence of this hyperparameter cannot easily be predicted - tuning it may possibly result in improved performance for both DP-GAN and RON-DP-GAN.

Exploring alternate utility measures is a logical extension given the reasoning presented in section 5.3. Alternate unsupervised machine learning tasks (for instance clustering) could be candidates. One could also consider training a classifier using the synthetic dataset as the inputs and the true labels as the targets. The accuracy of the classifier could then be used as a utility measure. However, adopting this approach would require a modification of DP-GAN to be based on a Conditional GAN as opposed to a pure GAN. This is necessary to be able to assign true labels to the generated synthetic samples.

Further experimentation to explore how DP-GAN performance depends on the prior used during training and sampling could also prove fruitful. The covariance matrix Σ_{DP} computed by RON-Gauss need not necessarily be symmetric positive semidefinite. Thus, Σ_{DP} is not truly the covariance matrix of a multivariate gaussian distribution. An interesting extension would be to train RON-DP-GAN after constraining Σ_{DP} to be symmetric positive semidefinite. Furthermore, RON-Gauss calculates μ_{DP} and Σ_{DP} under $(\varepsilon, 0)$-DP. One could explore a relaxation of RON-DP-GAN in which μ_{DP} and Σ_{DP} are calculated under (ε, δ)-DP. One might also explore using prior distributions defined by higher (private) moments of the underlying dataset (meaning moments beyond the first two).

In addition to the extensions described above, performing additional trials to see if empirically, utility increases as ε increases as expected intuitively would be useful. Testing the non-interactive private data release methods examined in this work on datasets other than MNIST would also be a useful extension. There is clear wealth of further inquiry that can be undertaken informed by the results of this work.

Bibliography

[1] Thee Chanyaswad, Changchang Liu, and Prateek Mittal. "Coupling Dimensionality Reduction with Generative Model for Non-Interactive Private Data Release". In: *CoRR* abs/1709.00054 (2017). arXiv: 1709.00054. URL: http://arxiv.org/abs/1709.00054.

[2] Cynthia Dwork. "Differential Privacy". In: *Automata, Languages and Programming*. Ed. by Michele Bugliesi et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–12. ISBN: 978-3-540-35908-1.

[3] Cynthia Dwork and Aaron Roth. "The Algorithmic Foundations of Differential Privacy". In: *Found. Trends Theor. Comput. Sci.* 9.3–4 (Aug. 2014), pp. 211–407. ISSN: 1551-305X. DOI: 10.1561/0400000042. URL: http://dx.doi.org/10.1561/0400000042.

[4] Ryan Singel. "Netflix Cancels Recommendation Contest". In: *Wired* (Mar. 12, 2010). URL: https://www.wired.com/2010/03/netflix-cancels-contest/.

[5] Arvind Narayanan and Vitaly Shmatikov. "Robust De-anonymization of Large Sparse Datasets". In: *Proceedings of the 2008 IEEE Symposium on Security and Privacy*. SP '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 111–125. ISBN: 978-0-7695-3168-7. DOI: 10.1109/SP.2008.33. URL: https://doi.org/10.1109/SP.2008.33.

[6] Martin Abadi et al. "Deep Learning with Differential Privacy". In: *23rd ACM Conference on Computer and Communications Security (ACM CCS)*. 2016, pp. 308–318. URL: https://arxiv.org/abs/1607.00133.

[7] Ilya Mironov. "Rényi Differential Privacy". In: *2017 IEEE 30th Computer Security Foundations Symposium (CSF)* (2017), pp. 263–275.

[8] Ian Goodfellow et al. "Generative Adversarial Nets". In: *Advances in Neural Information Processing Systems 27*. Ed. by Z. Ghahramani et al. Curran Associates, Inc., 2014, pp. 2672–2680. URL: http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf.

[9] G. Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of Control, Signals and Systems* 2.4 (Dec. 1989), pp. 303–314. ISSN: 1435-568X. DOI: 10.1007/BF02551274. URL: https://doi.org/10.1007/BF02551274.

[10] Faizan Shaikh. *Introductory guide to Generative Adversarial Networks (GANs) and their promise!* June 2015. URL: https://www.analyticsvidhya.com/blog/2017/06/introductory-generative-adversarial-networks-gans/.

[11] Naveen Kodali et al. "How to Train Your DRAGAN". In: *CoRR* abs/1705.07215 (2017). arXiv: 1705.07215. URL: http://arxiv.org/abs/1705.07215.

[12] Alec Radford, Luke Metz, and Soumith Chintala. "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks". In: *ICLR*. 2016.

[13] H. Brendan McMahan and Galen Andrew. "A General Approach to Adding Differential Privacy to Iterative Training Procedures". In: *CoRR* abs/1812.06210 (2018).

[14] Aleksei Triastcyn and Boi Faltings. *Generating Differentially Private Datasets Using GANs*. 2018. URL: https://openreview.net/forum?id=rJv4XWZA-.

[15] Liyang Xie et al. "Differentially Private Generative Adversarial Network". In: *CoRR* abs/1802.06739 (2018). arXiv: 1802.06739. URL: http://arxiv.org/abs/1802.06739.

[16] Xinyang Zhang, Shouling Ji, and Ting Wang. "Differentially Private Releasing via Deep Generative Model". In: *CoRR* abs/1801.01594 (2018). arXiv: 1801.01594. URL: http://arxiv.org/abs/1801.01594.

[17] znxlwm. *tensorflow-MNIST-GAN-DCGAN*. https://github.com/znxlwm/tensorflow-MNIST-GAN-DCGAN. 2017.

[18] H. Brendan McMahan and Galen Andrew. *TensorFlow Privacy*. https://github.com/tensorflow/privacy. 2019.

Appendices

Appendix A

RON-Gauss Code

```
from tensorflow.examples.tutorials.mnist import input_data
import numpy as np
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
import scipy as sp
from RON_Gauss.ron_gauss_modified import RON_Gauss
from sklearn.preprocessing import normalize
import warnings

# load MNIST
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True, reshape=[])
proxy_mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
proxy_train_set = (proxy_mnist.train.images - 0.5) / 0.5

# Get true first principle component
true_pca = PCA(n_components=1)
```

```
true_pca.fit(proxy_train_set)
true_subspace = true_pca.components_
true_component_1 = true_subspace[0,:]

# Function to perform experiments
def run_experiments(eps, trials):
    results = np.zeros(trials)

    for i in range(trials):
        # Generate synthetic dataset
        rongauss = RON_Gauss(algorithm='unsupervised', epsilonMean = 0.3*eps,
                             epsilonCov = 0.7*eps)
        sample_data, _, _, _ = rongauss.generate_dpdata(X=proxy_train_set, dimension=100, reconstruct=True, meanAdjusted=False)

        # Get first principle component of synthetic dataset
        synthetic_pca = PCA(n_components=1)
        synthetic_pca.fit(sample_data)
        synthetic_component = synthetic_pca.components_[0,:]
        results[i] = np.linalg.norm(true_component_1-synthetic_component)

    return(results)

# Perform experiments
epsilon = [1,1.5,2,2.5,3]
```

```
with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    data = np.zeros([5,50])
    for i in range(5):
        data[i,:] = run_experiments(eps=epsilon[i], trials=50)

# Function to plot results
def plot_results(iterations):
    values = np.zeros(5)
    for i in range(5):
        values[i] = np.mean(data[i,0:iterations])
    plt.scatter(epsilon, values)
    plt.xlabel('Epsilon')
    plt.title('Difference Between Synthetic Data First Principal'+
              'Component and Actual First Principal Component')
    plt.show()

# Plot results
plot_results(50)

# Save results
np.save('data',data)
```

Appendix B

DP-GAN Code

The code for DP-GAN is modelled from code in the tensorflow-MNIST-GAN-DCGAN GitHub repository [17].

```
import os, time, itertools, imageio, pickle, sys
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
from scipy.optimize import bisect
from privacy.analysis import privacy_ledger
from privacy.analysis.rdp_accountant import compute_rdp
from privacy.analysis.rdp_accountant import get_privacy_spent
from privacy.optimizers import dp_optimizer
from sklearn.decomposition import PCA

# Read inputs from command line
target_eps = sys.argv[1]
trial_num = sys.argv[2]
```

```

# Fixed training parameters
batch_size = 100
lr = 0.0002
train_epoch = 20
num_microbatches=50
l2_norm_clip=1.5
delta = 1e-5
dev = 0.5

# Calculate noise_multipler from input epsilon value
orders = [1 + x / 10. for x in range(1, 100)] + list(
    range(12, 64))

sampling_probability = batch_size / 55000
steps = train_epoch * 55000 // batch_size

def find_eps(multiplier):
    rdp = compute_rdp(q=sampling_probability,
                      noise_multiplier=multiplier,
                      steps=steps,
                      orders=orders)
    return(get_privacy_spent(orders, rdp, target_delta
        =delta)[0]-float(target_eps))

noise_multiplier = bisect(find_eps,0.5,3.0)

rdp = compute_rdp(q=sampling_probability,
                  noise_multiplier=noise_multiplier,
                  steps=steps,
                  orders=orders)
```

```

epsilon = get_privacy_spent(orders, rdp, target_delta=
    delta)[0]

# Leaky relu function definition
def lrelu(x, th=0.2):
    return tf.maximum(th * x, x)

# Generator Network
def generator(x, isTrain=True, reuse=False):
    with tf.variable_scope('generator', reuse=reuse):

        # 1st hidden layer
        conv1 = tf.layers.conv2d_transpose(x, 1024,
            [4, 4], strides=(1, 1), padding='valid')
        lrelu1 = lrelu(tf.layers.batch_normalization(
            conv1, training=isTrain), 0.2)

        # 2nd hidden layer
        conv2 = tf.layers.conv2d_transpose(lrelu1,
            512, [4, 4], strides=(2, 2), padding='same'
            )
        lrelu2 = lrelu(tf.layers.batch_normalization(
            conv2, training=isTrain), 0.2)

        # 3rd hidden layer
        conv3 = tf.layers.conv2d_transpose(lrelu2,
            256, [4, 4], strides=(2, 2), padding='same'
            )
        lrelu3 = lrelu(tf.layers.batch_normalization(
            conv3, training=isTrain), 0.2)
```

```

        # 4th hidden layer
        conv4 = tf.layers.conv2d_transpose(lrelu3, 128, [4, 4], strides=(2, 2), padding='same')
        lrelu4 = lrelu(tf.layers.batch_normalization(conv4, training=isTrain), 0.2)

        # output layer
        conv5 = tf.layers.conv2d_transpose(lrelu4, 1, [4, 4], strides=(2, 2), padding='same')
        o = tf.nn.tanh(conv5)

        return o

# Discriminator Network
def discriminator(x, isTrain=True, reuse=False):
    with tf.variable_scope('discriminator', reuse=reuse):
        # 1st hidden layer
        conv1 = tf.layers.conv2d(x, 128, [4, 4], strides=(2, 2), padding='same')
        lrelu1 = lrelu(conv1, 0.2)
        noise1 = tf.random_normal(shape=tf.shape(lrelu1),mean=0.0,stddev=dev)
        lrelu1 = tf.add(lrelu1,noise1)

        # 2nd hidden layer
        conv2 = tf.layers.conv2d(lrelu1, 256, [4, 4], strides=(2, 2), padding='same')
```

```
        lrelu2 = lrelu(tf.layers.batch_normalization(
            conv2, training=isTrain), 0.2)
        noise2 = tf.random_normal(shape=tf.shape(
            lrelu2),mean=0.0,stddev=dev)
        lrelu2 = tf.add(lrelu2,noise2)

        # 3rd hidden layer
        conv3 = tf.layers.conv2d(lrelu2, 512, [4, 4],
            strides=(2, 2), padding='same')
        lrelu3 = lrelu(tf.layers.batch_normalization(
            conv3, training=isTrain), 0.2)
        noise3 = tf.random_normal(shape=tf.shape(
            lrelu3),mean=0.0,stddev=dev)
        lrelu3 = tf.add(lrelu3,noise3)

        # 4th hidden layer
        conv4 = tf.layers.conv2d(lrelu3, 1024, [4, 4],
             strides=(2, 2), padding='same')
        lrelu4 = lrelu(tf.layers.batch_normalization(
            conv4, training=isTrain), 0.2)
        noise4 = tf.random_normal(shape=tf.shape(
            lrelu4),mean=0.0,stddev=dev)
        lrelu4 = tf.add(lrelu4,noise4)

        # output layer
        conv5 = tf.layers.conv2d(lrelu4, 1, [4, 4],
            strides=(1, 1), padding='valid')
        o = tf.nn.sigmoid(conv5)

        return o, conv5

```

```
# Function to display samples from generator
fixed_z_ = np.random.normal(0, 1, (25, 1, 1, 100))
def show_result(num_epoch, show = False, save = False,
    path = 'result.png'):
    test_images = sess.run(G_z, {z: fixed_z_, isTrain:
        False})

    size_figure_grid = 5
    fig, ax = plt.subplots(size_figure_grid,
        size_figure_grid, figsize=(5, 5))
    for i, j in itertools.product(range(
        size_figure_grid), range(size_figure_grid)):
        ax[i, j].get_xaxis().set_visible(False)
        ax[i, j].get_yaxis().set_visible(False)

    for k in range(size_figure_grid*size_figure_grid):
        i = k // size_figure_grid
        j = k % size_figure_grid
        ax[i, j].cla()
        ax[i, j].imshow(np.reshape(test_images[k],
            (64, 64)), cmap='gray')

    label = 'Epoch␣{0}'.format(num_epoch)
    fig.text(0.5, 0.04, label, ha='center')

    if save:
        plt.savefig(path)

    if show:
        plt.show()
    else:
```

```
        plt.close()

# Function to plot training progress
def show_train_hist(hist, show = False, save = False, path = 'Train_hist.png'):
    x = range(len(hist['D_losses']))

    y1 = hist['D_losses']
    y2 = hist['G_losses']

    plt.plot(x, y1, label='D_loss')
    plt.plot(x, y2, label='G_loss')

    plt.xlabel('Epoch')
    plt.ylabel('Loss')

    plt.legend(loc=4)
    plt.grid(True)
    plt.tight_layout()

    if save:
        plt.savefig(path)

    if show:
        plt.show()
    else:
        plt.close()

# Load MNIST
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True, reshape=[])
```

```
# Variables : input
x = tf.placeholder(tf.float32, shape=(None, 64, 64, 1))
z = tf.placeholder(tf.float32, shape=(None, 1, 1, 100))
isTrain = tf.placeholder(dtype=tf.bool)

# Networks : generator
G_z = generator(z, isTrain)

# networks : discriminator
D_real, D_real_logits = discriminator(x, isTrain)
D_fake, D_fake_logits = discriminator(G_z, isTrain, reuse=True)

# Loss for each network
D_loss_real = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=D_real_logits, labels=tf.ones([batch_size, 1, 1, 1])))
D_loss_fake = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=D_fake_logits, labels=tf.zeros([batch_size, 1, 1, 1])))
D_loss = D_loss_real+D_loss_fake

vector_G_loss = tf.nn.sigmoid_cross_entropy_with_logits(logits=D_fake_logits, labels=tf.ones([batch_size, 1, 1, 1]))
```

```
G_loss = tf.reduce_mean(vector_G_loss)

# Trainable variables for each network
T_vars = tf.trainable_variables()
D_vars = [var for var in T_vars if var.name.startswith('discriminator')]
G_vars = [var for var in T_vars if var.name.startswith('generator')]

# Define DP optimizer
ledger = privacy_ledger.PrivacyLedger(
                population_size=55000,
                selection_probability=(batch_size/55000),
                max_samples=1e6,
                max_queries=1e6)

G_optimizer = dp_optimizer.DPAdamGaussianOptimizer(
            l2_norm_clip=l2_norm_clip,
            noise_multiplier=noise_multiplier,
            num_microbatches=num_microbatches,
            learning_rate=lr,
            beta1=0.5,
            ledger=ledger)

# Pptimizer for each network
with tf.control_dependencies(tf.get_collection(tf.GraphKeys.UPDATE_OPS)):
    D_optim = tf.train.AdamOptimizer(lr, beta1=0.5).minimize(D_loss, var_list=D_vars)
    G_optim= G_optimizer.minimize(loss=vector_G_loss, var_list=G_vars)
```

```python


# Open session and initialize all variables
saver = tf.train.Saver()
sess = tf.InteractiveSession()

tf.global_variables_initializer().run()

# MNIST resize and normalization
train_set = tf.image.resize_images(mnist.train.images,
    [64, 64]).eval()
train_set = (train_set - 0.5) / 0.5  # normalization;
    range: -1 ~ 1

# Results save folder
root = 'Eps_'+str(target_eps)+'_Trial'+str(trial_num)+
    '_Private_MNIST_DCGAN_results/'
model = 'Private_MNIST_DCGAN_'
if not os.path.isdir(root):
    os.mkdir(root)
if not os.path.isdir(root + 'Fixed_results'):
    os.mkdir(root + 'Fixed_results')

train_hist = {}
train_hist['D_losses'] = []
train_hist['G_losses'] = []
train_hist['per_epoch_ptimes'] = []
train_hist['total_ptime'] = []

# Training-loop
np.random.seed(int(time.time()))
```

```
print('training start!')
start_time = time.time()
for epoch in range(train_epoch):
    G_losses = []
    D_losses = []
    epoch_start_time = time.time()
    for iter in range(mnist.train.num_examples // batch_size):
        # update discriminator
        x_ = train_set[iter*batch_size:(iter+1)*batch_size]
        z_ = np.random.normal(0, 1, (batch_size, 1, 1, 100))

        loss_d_, _ = sess.run([D_loss, D_optim], {x: x_, z: z_, isTrain: True})
        D_losses.append(loss_d_)

        # update generator
        z_ = np.random.normal(0, 1, (batch_size, 1, 1, 100))
        loss_g_, _ = sess.run([vector_G_loss, G_optim], {z: z_, x: x_, isTrain: True})
        G_losses.append(loss_g_)

    epoch_end_time = time.time()
    per_epoch_ptime = epoch_end_time - epoch_start_time
    print('[%d/%d] - ptime: %.2f loss_d: %.3f, loss_g: %.3f' % ((epoch + 1), train_epoch,
```

```python
            per_epoch_ptime, np.mean(D_losses), np.mean(G_losses)))
        fixed_p = root + 'Fixed_results/' + model + str(epoch + 1) + '.png'
        show_result((epoch + 1), save=True, path=fixed_p)
        train_hist['D_losses'].append(np.mean(D_losses))
        train_hist['G_losses'].append(np.mean(G_losses))
        train_hist['per_epoch_ptimes'].append(per_epoch_ptime)

end_time = time.time()
total_ptime = end_time - start_time
train_hist['total_ptime'].append(total_ptime)

print('Avg per epoch ptime: %.2f, total %d epochs ptime: %.2f' % (np.mean(train_hist['per_epoch_ptimes']), train_epoch, total_ptime))
print("Training finish!... save training results")
with open(root + model + 'train_hist.pkl', 'wb') as f:
    pickle.dump(train_hist, f)

show_train_hist(train_hist, save=True, path=root + model + 'train_hist.png')

images = []
for e in range(train_epoch):
    img_name = root + 'Fixed_results/' + model + str(e + 1) + '.png'
    images.append(imageio.imread(img_name))
imageio.mimsave(root + model + 'generation_animation.gif', images, fps=5)
```

```

# Save model
save_path = saver.save(sess, root+"model.ckpt")
print("Model saved in path: %s" % save_path)

train_set = np.resize(train_set,[55000,64*64])

# Get true first principle component
true_pca = PCA(n_components=20)
true_pca.fit(train_set)
true_subspace = true_pca.components_
true_component_1 = true_subspace[0,:]

# Function to create synthetic dataset from generator
def generate_sample(size):
    z_ = np.random.normal(0,1,(size,1,1,100))
    test_images = sess.run(G_z, {z: z_, isTrain: False
        })
    return np.resize(test_images,[size,64*64])

def calc_mean(array):
    mean = np.mean(array)
    sd = np.std(array)
    interval = np.zeros(2)
    interval[0] = mean-1.96*(sd/np.sqrt(len(array)))
    interval[1] = mean+1.96*(sd/np.sqrt(len(array)))
    return(mean,interval)

def evaluate_performance(trials):

    prim_dist = np.zeros(trials)
```

```

        for i in range(trials):

            # Draw a fresh sample from the model and perform PCA
            sample = np.zeros([55000,64*64])

            for j in range(550):
                sample[0+100*j:100+100*j,:] = generate_sample(100)

            synthetic_pca = PCA(n_components=20)
            synthetic_pca.fit(sample)
            synthetic_subspace = synthetic_pca.components_

            # Calc distance between the first principal components
            prim_dist[i] = np.linalg.norm(true_component_1-synthetic_subspace[0,:])

        prim_mean, prim_interval = calc_mean(array=prim_dist)

        return(prim_mean,prim_interval)

print('Evaluation Start!')
start_time = time.time()
a, b= evaluate_performance(10)
end_time = time.time()
elapsed_time = end_time - start_time
print('ptime: %.2f' % (elapsed_time))
```

```
print('The mean distance between the first principal component of the two subspaces is: ' + str(a))
print('The confidence interval of this mean is: ' + str(b))
print('epsilon = ' + str(epsilon) + '\ndelta = ' + str(delta) +
        '\nnoise_multiplier = ' + str(noise_multiplier))
sess.close()
```

Appendix C

RON-DP-GAN Code

The code for RON-DP-GAN is modelled from code in the tensorflow-MNIST-GAN-DCGAN GitHub repository [17].

```
import os, time, itertools, imageio, pickle, sys
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
from scipy.optimize import bisect
from privacy.analysis import privacy_ledger
from privacy.analysis.rdp_accountant import compute_rdp
from privacy.analysis.rdp_accountant import get_privacy_spent
from privacy.optimizers import dp_optimizer
from RON_Gauss.ron_gauss_modified import RON_Gauss
from sklearn.decomposition import PCA

# Read inputs from command line
target_eps = sys.argv[1]
```

```
eps_multiplier = sys.argv[2]
trial_num = sys.argv[3]

# Fixed training parameters
batch_size = 100
lr = 0.0002
train_epoch = 20
l2_norm_clip=1.5
num_microbatches=50
delta = 1e-5
dev = 0.5

# Determine epsilon values for RON-Gauss and DP-GAN
ratio = float(eps_multiplier)
epsRON = ratio*float(target_eps)
epsGAN = (1-ratio)*float(target_eps)

# Calculate noise_multipler from input epsilon value
orders = [1 + x / 10. for x in range(1, 100)] + list(
    range(12, 64))

sampling_probability = batch_size / 55000
steps = train_epoch * 55000 // batch_size

def find_eps(multiplier):
    rdp = compute_rdp(q=sampling_probability,
                      noise_multiplier=multiplier,
                      steps=steps,
                      orders=orders)
    return(get_privacy_spent(orders, rdp, target_delta
        =delta)[0]-epsGAN)
```

```python

noise_multiplier = bisect(find_eps,0.5,3)

rdp = compute_rdp(q=sampling_probability,
                  noise_multiplier=noise_multiplier,
                  steps=steps,
                  orders=orders)

epsilon = get_privacy_spent(orders, rdp, target_delta=delta)[0]

# Leaky relu function definition
def lrelu(x, th=0.2):
    return tf.maximum(th * x, x)

# Generator Network
def generator(x, isTrain=True, reuse=False):
    with tf.variable_scope('generator', reuse=reuse):

        # 1st hidden layer
        conv1 = tf.layers.conv2d_transpose(x, 1024, [4, 4], strides=(1, 1), padding='valid')
        lrelu1 = lrelu(tf.layers.batch_normalization(conv1, training=isTrain), 0.2)

        # 2nd hidden layer
        conv2 = tf.layers.conv2d_transpose(lrelu1, 512, [4, 4], strides=(2, 2), padding='same')
        lrelu2 = lrelu(tf.layers.batch_normalization(conv2, training=isTrain), 0.2)
```

```
        # 3rd hidden layer
        conv3 = tf.layers.conv2d_transpose(lrelu2,
            256, [4, 4], strides=(2, 2), padding='same'
            )
        lrelu3 = lrelu(tf.layers.batch_normalization(
            conv3, training=isTrain), 0.2)

        # 4th hidden layer
        conv4 = tf.layers.conv2d_transpose(lrelu3,
            128, [4, 4], strides=(2, 2), padding='same'
            )
        lrelu4 = lrelu(tf.layers.batch_normalization(
            conv4, training=isTrain), 0.2)

        # output layer
        conv5 = tf.layers.conv2d_transpose(lrelu4, 1,
            [4, 4], strides=(2, 2), padding='same')
        o = tf.nn.tanh(conv5)

        return o

# Discriminator Network
def discriminator(x, isTrain=True, reuse=False):
    with tf.variable_scope('discriminator', reuse=
        reuse):
        # 1st hidden layer
        conv1 = tf.layers.conv2d(x, 128, [4, 4],
            strides=(2, 2), padding='same')
        lrelu1 = lrelu(conv1, 0.2)
```

```
        noise1 = tf.random_normal(shape=tf.shape(
            lrelu1),mean=0.0,stddev=dev)
        lrelu1 = tf.add(lrelu1,noise1)

        # 2nd hidden layer
        conv2 = tf.layers.conv2d(lrelu1, 256, [4, 4],
            strides=(2, 2), padding='same')
        lrelu2 = lrelu(tf.layers.batch_normalization(
            conv2, training=isTrain), 0.2)
        noise2 = tf.random_normal(shape=tf.shape(
            lrelu2),mean=0.0,stddev=dev)
        lrelu2 = tf.add(lrelu2,noise2)

        # 3rd hidden layer
        conv3 = tf.layers.conv2d(lrelu2, 512, [4, 4],
            strides=(2, 2), padding='same')
        lrelu3 = lrelu(tf.layers.batch_normalization(
            conv3, training=isTrain), 0.2)
        noise3 = tf.random_normal(shape=tf.shape(
            lrelu3),mean=0.0,stddev=dev)
        lrelu3 = tf.add(lrelu3,noise3)

        # 4th hidden layer
        conv4 = tf.layers.conv2d(lrelu3, 1024, [4, 4],
             strides=(2, 2), padding='same')
        lrelu4 = lrelu(tf.layers.batch_normalization(
            conv4, training=isTrain), 0.2)
        noise4 = tf.random_normal(shape=tf.shape(
            lrelu4),mean=0.0,stddev=dev)
        lrelu4 = tf.add(lrelu4,noise4)

```

```
        # output layer
        conv5 = tf.layers.conv2d(lrelu4, 1, [4, 4], strides=(1, 1), padding='valid')
        o = tf.nn.sigmoid(conv5)

        return o, conv5

# load MNIST
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True, reshape=[])

# Function to display samples from generator
def show_result(num_epoch, show = False, save = False, path = 'result.png'):
    test_images = sess.run(G_z, {z: fixed_z_, isTrain: False})

    size_figure_grid = 5
    fig, ax = plt.subplots(size_figure_grid, size_figure_grid, figsize=(5, 5))
    for i, j in itertools.product(range(size_figure_grid), range(size_figure_grid)):
        ax[i, j].get_xaxis().set_visible(False)
        ax[i, j].get_yaxis().set_visible(False)

    for k in range(size_figure_grid*size_figure_grid):
        i = k // size_figure_grid
        j = k % size_figure_grid
        ax[i, j].cla()
        ax[i, j].imshow(np.reshape(test_images[k], (64, 64)), cmap='gray')
```

```python

    label = 'Epoch {0}'.format(num_epoch)
    fig.text(0.5, 0.04, label, ha='center')

    if save:
        plt.savefig(path)

    if show:
        plt.show()
    else:
        plt.close()

# Function to plot training progress
def show_train_hist(hist, show = False, save = False, path = 'Train_hist.png'):
    x = range(len(hist['D_losses']))

    y1 = hist['D_losses']
    y2 = hist['G_losses']

    plt.plot(x, y1, label='D_loss')
    plt.plot(x, y2, label='G_loss')

    plt.xlabel('Epoch')
    plt.ylabel('Loss')

    plt.legend(loc=4)
    plt.grid(True)
    plt.tight_layout()

    if save:
```

```
            plt.savefig(path)

        if show:
            plt.show()
        else:
            plt.close()

# Variables : input
x = tf.placeholder(tf.float32, shape=(None, 64, 64, 1))
z = tf.placeholder(tf.float32, shape=(None, 1, 1, 100))
isTrain = tf.placeholder(dtype=tf.bool)

# Networks : generator
G_z = generator(z, isTrain)

# Networks : discriminator
D_real, D_real_logits = discriminator(x, isTrain)
D_fake, D_fake_logits = discriminator(G_z, isTrain, reuse=True)

# Loss for each network
D_loss_real = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=D_real_logits, labels=tf.ones([batch_size, 1, 1, 1])))
D_loss_fake = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=D_fake_logits, labels=tf.zeros([batch_size, 1, 1, 1])))
```

```
D_loss = D_loss_real + D_loss_fake
vector_G_loss = tf.nn.
    sigmoid_cross_entropy_with_logits(logits=
    D_fake_logits, labels=tf.ones([batch_size, 1, 1,
    1]))
G_loss = tf.reduce_mean(vector_G_loss)

# Trainable variables for each network
T_vars = tf.trainable_variables()
D_vars = [var for var in T_vars if var.name.startswith
    ('discriminator')]
G_vars = [var for var in T_vars if var.name.startswith
    ('generator')]

# Define DP optimizer
ledger = privacy_ledger.PrivacyLedger(
      population_size=55000,
      selection_probability=(batch_size/55000),
      max_samples=1e6,
      max_queries=1e6)

optimizer = dp_optimizer.DPAdamGaussianOptimizer(
      l2_norm_clip=l2_norm_clip,
      noise_multiplier=noise_multiplier,
      num_microbatches=num_microbatches,
      learning_rate=lr,
      beta1=0.5,
      ledger=ledger)

# Optimizer for each network
```

```
with tf.control_dependencies(tf.get_collection(tf.
    GraphKeys.UPDATE_OPS)):
      D_optim = tf.train.AdamOptimizer(lr, beta1=0.5).
          minimize(D_loss, var_list=D_vars)
      G_optim= optimizer.minimize(loss=vector_G_loss,
          var_list=G_vars)

# Open session and initialize all variables
saver = tf.train.Saver()
sess = tf.InteractiveSession()

tf.global_variables_initializer().run()

# MNIST resize and normalization
train_set = tf.image.resize_images(mnist.train.images,
     [64, 64]).eval()
train_set = (train_set - 0.5) / 0.5

proxy_mnist = input_data.read_data_sets("MNIST_data/",
     one_hot=True)
proxy_train_set = (proxy_mnist.train.images - 0.5) /
    0.5

# Use RON-Gauss to find DP mean and covariance
rongauss = RON_Gauss(algorithm='unsupervised',
    epsilonMean = 0.3*epsRON, epsilonCov = 0.7*epsRON)
_, _, mu, var = rongauss.generate_dpdata(X=
    proxy_train_set, dimension=100, reconstruct=False,
    meanAdjusted=False)
fixed_z_ = np.random.multivariate_normal(mu, var,
    (25,1,1))
```

```

# Results save folder
root = 'Eps_'+str(target_eps)+'_Trial'+str(trial_num)+'_RON_Private_MNIST_DCGAN_results/'
model = 'RON_Private_MNIST_DCGAN_'
if not os.path.isdir(root):
    os.mkdir(root)
if not os.path.isdir(root + 'Fixed_results'):
    os.mkdir(root + 'Fixed_results')

train_hist = {}
train_hist['D_losses'] = []
train_hist['G_losses'] = []
train_hist['per_epoch_ptimes'] = []
train_hist['total_ptime'] = []

# Training-loop
np.random.seed(int(time.time()))
print('training start!')
start_time = time.time()
for epoch in range(train_epoch):
    G_losses = []
    D_losses = []
    epoch_start_time = time.time()
    for iter in range(mnist.train.num_examples // batch_size):
        # update discriminator
        x_ = train_set[iter*batch_size:(iter+1)*batch_size]
        z_ = np.random.multivariate_normal(mu, var, (batch_size,1,1))
```

```

        loss_d_, _ = sess.run([D_loss, D_optim], {x: x_, z: z_, isTrain: True})
        D_losses.append(loss_d_)

        # update generator
        z_ = np.random.multivariate_normal(mu, var, (batch_size,1,1))
        loss_g_, _ = sess.run([vector_G_loss, G_optim], {z: z_, x: x_, isTrain: True})
        G_losses.append(loss_g_)

    epoch_end_time = time.time()
    per_epoch_ptime = epoch_end_time - epoch_start_time
    print('[%d/%d] - ptime: %.2f loss_d: %.3f, loss_g: %.3f' % ((epoch + 1), train_epoch, per_epoch_ptime, np.mean(D_losses), np.mean(G_losses)))
    fixed_p = root + 'Fixed_results/' + model + str(epoch + 1) + '.png'
    show_result((epoch + 1), save=True, path=fixed_p)
    train_hist['D_losses'].append(np.mean(D_losses))
    train_hist['G_losses'].append(np.mean(G_losses))
    train_hist['per_epoch_ptimes'].append(per_epoch_ptime)

end_time = time.time()
total_ptime = end_time - start_time
train_hist['total_ptime'].append(total_ptime)

```

```
print('Avg per epoch ptime: %.2f, total %d epochs ptime: %.2f' % (np.mean(train_hist['per_epoch_ptimes']), train_epoch, total_ptime))
print("Training finish!... save training results")
with open(root + model + 'train_hist.pkl', 'wb') as f:
    pickle.dump(train_hist, f)

show_train_hist(train_hist, save=True, path=root + model + 'train_hist.png')

images = []
for e in range(train_epoch):
    img_name = root + 'Fixed_results/' + model + str(e + 1) + '.png'
    images.append(imageio.imread(img_name))
imageio.mimsave(root + model + 'generation_animation.gif', images, fps=5)

# Save model
save_path = saver.save(sess, root+"model.ckpt")
print("Model saved in path: %s" % save_path)

np.save(root+'RON_mu.npy',mu)
np.save(root+'RON_var.npy',var)

train_set = np.resize(train_set,[55000,64*64])

# Get true first principle component
true_pca = PCA(n_components=20)
true_pca.fit(train_set)
true_subspace = true_pca.components_
```

```
true_component_1 = true_subspace[0,:]
true_component_2 = true_subspace[1,:]
true_component_3 = true_subspace[2,:]

# Function to create synthetic dataset from generator
def generate_sample(size):
    z_ = np.random.normal(0,1,(size,1,1,100))
    test_images = sess.run(G_z, {z: z_, isTrain: False})
    return np.resize(test_images,[size,64*64])

def calc_mean(array):
    mean = np.mean(array)
    sd = np.std(array)
    interval = np.zeros(2)
    interval[0] = mean-1.96*(sd/np.sqrt(len(array)))
    interval[1] = mean+1.96*(sd/np.sqrt(len(array)))
    return(mean,interval)

def evaluate_performance(trials):

    prim_dist = np.zeros(trials)

    for i in range(trials):

        # Draw a fresh sample from the model and perform PCA
        sample = np.zeros([55000,64*64])

        for j in range(550):
```

```
                sample[0+100*j:100+100*j,:] = generate_sample(100)

            synthetic_pca = PCA(n_components=20)
            synthetic_pca.fit(sample)
            synthetic_subspace = synthetic_pca.components_

            # Calc distance between the first 3 principal components
            prim_dist[i] = np.linalg.norm(true_component_1-synthetic_subspace[0,:])

        prim_mean, prim_interval = calc_mean(array=prim_dist)

        return(prim_mean,prim_interval)

print('Evaluation Start!')
start_time = time.time()
a, b = evaluate_performance(10)
end_time = time.time()
elapsed_time = end_time - start_time
print('ptime: %.2f' % (elapsed_time))
print('The mean distance between the first principal component of the two subspaces is: ' + str(a))
print('The confidence interval of this mean is: ' + str(b))
print('total epsilon = ' + str(epsilon+epsRON) + '\ndelta = ' + str(delta) +
        '\nnoise multiplier = ' + str(noise_multiplier)
      +'\nRON epsilon = ' + str(epsRON)+
```

```
        '\nGAN epsilon = ' + str(epsilon)+
        'Eps ratio = ' + ratio)
sess.close()
```