One-Class Support Vector Machine Generative Adversarial Network

by

Siting Wang

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Statistical Machine Learning

Department of Mathematics and Statistics
University of Alberta

Abstract

Generative Adversarial Networks (GAN) is a field of popular generating models, and there are many variants these years. Lim and Ye 2017 proposed the Geometric GAN to connect the network with the geometric interpretation. They update the discriminator based on the algorithm of Support Vector Machines (SVM),

Inspired from their work, we proposed a new algorithm using the robust One Class Support Vector Machines. We also proposed that the discriminator should separate the dataset into three groups: the correctly classified real data, the correctly classified generated data, and the incorrectly classified data. By eliminating the space of incorrectly classified data, we can have our discriminator capture more patterns. We tested our model and the Geometric GAN on the MNIST dataset, and our model has better performance on the same setting.

Preface

This thesis is an original work by Siting Wang. No part of this thesis has been previously published.

Table of Contents

1 Introduction **1**

1.1 Motivation . . . 2

1.2 Thesis Objectives . . . 3

1.3 Thesis Outline . . . 3

2 Background **4**

2.1 Generative Adversarial Network . . . 4

2.1.1 Adversarial Nets . . . 4

2.1.2 Mean Feature GAN . . . 6

2.1.3 Geometric GAN . . . 8

2.2 One Class Support Vector Machine . . . 11

2.2.1 Support Vector Machine . . . 11

2.2.2 One-Class SVM . . . 14

2.3 Gap in Research . . . 16

2.4 Conclusions . . . 16

3 Methods and Procedure **18**

3.1 One-Class SVM Hyperplane . . . 18

3.1.1 Updating the Unsupervised Network . . . 20

3.2 Experiment & Tuning . . . 21

3.2.1 Network . . . 24

3.3 Results and Discussion . . . 24

3.4 Conclusions 26

4 Conclusions, Recommendations, & Future Work **28**

4.1 Conclusions 28

4.2 Future Work 28

Bibliography **30**

Appendix A: Related Networks **32**

A.1 DC-GAN 32

Appendix B: Experiment Codes **34**

B.1 OCSVM GAN 34

Appendix C: Simulation Comparison **53**

List of Tables

3.1 The Labels from Different Class of Data to Different Classifier 19

List of Figures

2.1 Structure of GAN 5

2.2 From both plots, we can see that both methods creates a linear classifier to discriminate the dataset. (Lim and Ye 2017) 9

3.1 The implication of our discriminator. We have red and blue separating lines. The red and blue area are the area of inliers for each classifier. Then the red and blue dots are correctly classified by both classifiers, while the purple and black dots are the inliers for both separators and outliers for both separators respectively. 19

3.2 Discriminator of OCSVM-GAN. 20

3.3 The Indication Plot of OCSVM: The red straight line is the separating hyperplane, and the red area is the area of inliers. All the round points are the inliers in our sample data. The green dashed line is the soft margin with respect to the the given margin distance. All the blue points will not affect the final result of the classification while all the green points are the support vectors. 22

3.4 The example images from MNIST Dataset 23

3.5 The Architecture of Network in Our Experiments 25

3.6 As shown in the plots, out model converges faster than the Geometric GAN. 26

3.7 In this plot, we can see that the Geometric does not any useful information. While our model has generated digits. 26

A.1 The Generator Architecture of DCGAN (Radford *et al.* 2015) 33

C.1 The Distribution of Simulation Training Data 54

C.2 The Distribution Generated Data from original GAN 54

C.3 The Distribution of Generated Data from W-GAN 55

C.4 The Distribution of Generated Data from Geometric GAN 55

C.5 The Distribution of Generated Data from OCSVMGAN 56

Chapter 1

Introduction

Generative models are the only models that can provide new data instead of the density functions only (Goodfellow 2016). It is also an area with potential of application. In practice, using generative models can effectively solve the problem of lacking samples. It can also help to create a safe virtual experiment environment. Using the Generative model can also help to avoid confidentiality dilemma. Besides that, the generative models can create many attractive applications.The most famous example is the Deepfake with praise and blames. In 2014, Goodfellow *et al.* 2014 invented the Generative Adversarial Net (GAN). It is a different model from the previous generative models.

Most of the generative models fall into the category of *Maximum Likelihood* or can be viewed as a Maximum Likelihood Problem (Goodfellow 2016). Generative Adversarial Network is a minimax problem, while we can also view it as a maximum likelihood problem. In Maximum Likelihood related generative models, we have the *explicit density* and *implicit density* models. The popular generative models Variational Autoencoder (VAE) is the explicit density model, and the Generative Adversarial Net is the implicit density model. Both VAEs and GANs are models difficult to optimize. The GANs are asymptotically consistent (Goodfellow 2016). Then is is more consistent. It also generate good quality results. However, GANs needs to reach the Nash equilibrium due to the property of minimax problem. We might take more time and effort on training the GANs even on a successful algorithm involving adversarial algorithms.

1.1 Motivation

Lim and Ye 2017 proposed the network called Geometric GAN. The network illustrated the discriminator of GAN with respect to Geometry. They proposed to use the Support Vector Machine (SVM) as the discriminator. By applying the Hinge Loss connected with the SVM, they created a more robust result in this optimization problem. Lim and Ye also tested on the simulation data. The Geometric GAN outperformed the other GAN variants, e.g. W-GAN (Arjovsky *et al.* 2017). Especially, the SVM network helps to eliminate the effect of the outliers, including the outliers from real dataset and generated dataset. Since we generally deal with high dimensional data in Generating model, we will see the generating network converges to a subspace that does not capture all the necessary features of real dataset, e.g mode collaspes. This affects both the discriminator and the generator from updating.

Inspired from the geometric interpretation of Geometric GAN , we realize that most of the discriminator from GANs only tries to enlarge the distance between the cluster of Real data with Generated data. The generators are responsible to 'cheat' the discriminator. While we notice that with high dimensional data, the discriminator cannot catch all the necessary patterns from the real data, and we can have the generated data not close to the real data in some dimensions. We can see the simulation evidence from the Chapter C.

We can solve this problem with deeper network, but with more complex data, we need deeper network to improve the results, and that makes the learning process more difficult.

1.2 Thesis Objectives

In this thesis, we use the One-Class Support Vector Machine (OCSVM) to be the classifier instead of the traditional Binary Support Vector Machine, and determine how to implant this unsupervised method into a labelled classification properly.

We also combined two OCSVM classifiers into one discriminator model, to generate a more robust model with the shallower network.

1.3 Thesis Outline

The thesis contains two main parts. Chapter 2 will introduce several kinds of Generative Adversarial Network in details. This is followed by a review of Support Vector Machines and One-Class Support Vector Machines. Chapter 3 will mainly discuss about the model we proposed. We will first introduce the algorithm. Then we have briefly discussed about the experiment result and the comparison with the Geometric GAN (Lim and Ye 2017). By the end we will discuss about the possible future works.

Chapter 2

Background

2.1 Generative Adversarial Network

Goodfellow *et al.* 2014 proposes a different framework for generative models. This network combines a discriminative model D to maximize the difference between the source data and generated data. GANs (Generative Adversarial Network) have a generative model G and a descriminative model D. To optimize the network, we apply a minimax operation to the loss function. It will update model G and D simultaneously. In general, the Generative model G learns the pattern of training dataset and generate the sample datasets with those patterns. The Discriminative model D classifies real data from the generated data. Two networks will compete with each other until the model converges and the generated data achieves our requirement.

2.1.1 Adversarial Nets

To begin with the adversarial networks, we first define inputs and their distributions.

Definition 1 *To avoid confusion, let's define source input data as* I

- $X : (x_1, \cdots, x_n) \sim \mathbb{P}_r$*: the input data that we want to generate follows a distribution* p_r*,* r *here indicates the real data.*

- $Z : (z_1, \cdots, z_m) \sim \mathbb{P}_z$ *: the random noise with sample number* m*.*

Definition 2 *We define two mapping models G and D. Both models are differentiable deep networks.*

- $G(z;\theta_g)$ *is the model of generator distribution* $\mathbb{P}_g$ *with parameters* θ_g. *While* $\mathbb{P}_g$ *is the predicted distribution of* $\mathbb{P}_r$.
- $D(x;\theta_d)$ *is a classification model to classify the input x from distribution X or $G(z)$. We optimize $D(x;\theta_d)$ to maximize the probability of correct classification.*

In practice, we consider $m = n$ in the following Theorem and definitions.

In original GAN (Goodfellow *et al.* 2014), the optimal discriminator D is

$$D_G^*(x) = \frac{p_{data}(x)}{p_{data}(x) + p_g(x)} \tag{2.1}$$

While x could be real input data X or generated data $G(Z)$.

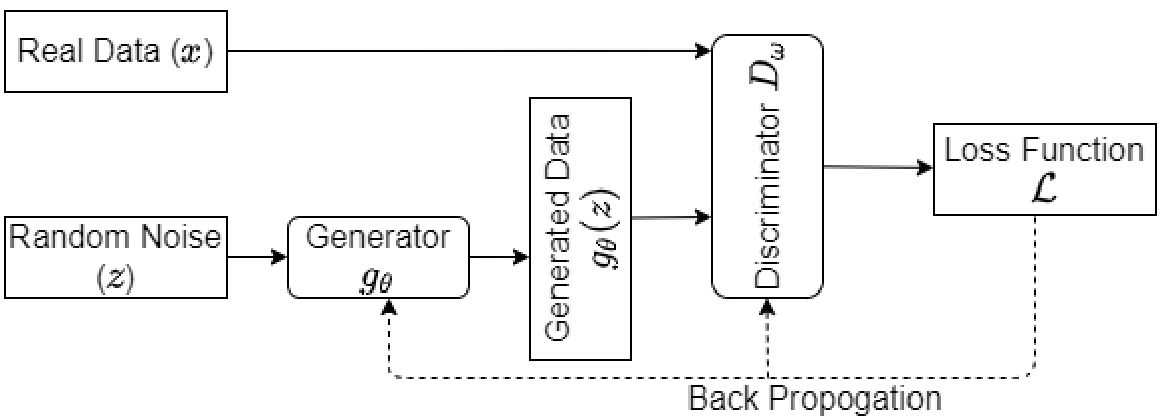


Figure 2.1: Structure of GAN

The minimax process with the value function $V(G, D)$:

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))] \tag{2.2}$$

In deep network, it is not feasible to find the global optimization. Gradient Descent is a population method to find a local optimization.

Algorithm 1 SGD (Stochastic Gradient Descent) of Generative Adversarial Nets. We have sample m for both training data and generated data. We choose a least expensive choices to update the Discriminator. Here η is the learning rate for updating the parameters. Any gradient-based learning rules works for this algorithm.

for Numbers of iterations **do**

Sample $\{z_1,\ z_2, \cdots, z_m\}$ from noise prior $p_g(z)$.

Sample m examples $\{x_1, \cdots, x_m\}$ from generating distribution set $p_{data}(x)$.

Update the Discriminator

$$\theta_d \leftarrow \theta_d + \eta \nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} [\log D(x_i) + \log(1 - D(G(z_i)))]$$

Sample m noise samples $\{z_1, \cdots, z_m\}$from noise prior $p_g(z)$

Update the Generator

$$\theta_g \leftarrow \theta_g + \eta \nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log(1 - D(G(z_i)))$$

end for

2.1.2 Mean Feature GAN

Mroueh *et al.* 2017 introduced the Integral Probability Metrics to maximize the discriminates between the distributions of real data and generated data.

Definition 3 (Integral Probability Metric) *Given* $(\mathbb{R}^d, \mathcal{X})$ *be a compact measurable space.* $\mathscr{F}$ *be a set where* $\forall f \in \mathscr{F}$, f *is a bounded measurable functions on* $\mathcal{X}$. $\mathscr{P}$ *is a set of probability measure on* $\mathcal{X}$. *For two probability distributions* $\mathbb{P}, \mathbb{Q} \in \mathscr{P}(\mathcal{X})$. *We define integral probability metric (IPM) to be:*

$$d_{\mathscr{F}}(\mathbb{P}, \mathbb{Q}) = \sup_{f \in \mathscr{F}} |\mathbb{E}_{x \sim \mathbb{P}} f(x) - \mathbb{E}_{x \sim \mathbb{Q}} f(x)|$$

For IPM we define above, $d_{\mathscr{F}}(\mathbb{P}, \mathbb{P}) = 0$, but for any two distribution $\mathbb{P}, \mathbb{Q}$, $d_{\mathscr{F}}(\mathbb{P}, \mathbb{Q}) = 0$ does not implie $\mathbb{P} = \mathbb{Q}$, and $d_{\mathscr{F}}$ is nonnegative. Therefore, $d_{\mathscr{F}}$ is a pseudo metric. Also in our definition, both $\mathbb{P}$ and $\mathbb{Q}$ are probability measure, we have $d_{\mathscr{F}}(\mathbb{P}, \mathbb{Q})$ symmetric. In a generative model, we define $\mathbb{P}_r \in \mathscr{P}(\mathcal{X})$, and $\mathbb{P}_\theta$ is the distribution of $g_\theta(z),\ z \sim p_z$ We then solve the minimax problem:

$$\min_{g_\theta} d_{\mathscr{F}}(\mathbb{P}_r, \mathbb{P}_\theta)$$

We use IPM to define Mean Feature Matching GAN. Define $\mathscr{F} :=< \omega, \Phi_\zeta(x) >$, where $\omega \in \mathbb{R}^M$ and $\Phi_\zeta : \mathcal{X} \to \mathbb{R}^M$ a non-linear feature map. Here M is unrelated with the sample number m.

Then we define the function space:

$$\mathscr{F}_{\omega,\zeta,p} = \{f(x) =< \omega, \Phi_\zeta(x) > | \omega \in \mathbb{R}^M, \|\omega\|_p \leq 1, \Phi_\zeta(x) : \mathcal{X} \to \mathbb{R}^M,\ \zeta \in \Xi\}$$

Combine the function space with the IPM, we get:

$$\begin{aligned} d_{\mathscr{F}_{\omega,\zeta,p}}(\mathbb{P}, \mathbb{Q}) &= \max_{\zeta \in \Xi, \omega \in \mathbb{R}^M, \|\omega\|_p \leq 1} \langle \omega, \mathbb{E}_{x\sim\mathbb{P}}\Phi_\zeta(x) - \mathbb{E}_{x\sim\mathbb{Q}}\Phi_\zeta(x)\rangle \\ &= \max_{\zeta \in \Xi} \left[\max_{\omega \in \mathbb{R}^M, \|\omega\|_p \leq 1} \langle \omega, \mathbb{E}_{x\sim\mathbb{P}}\Phi_\zeta(x) - \mathbb{E}_{x\sim\mathbb{Q}}\Phi_\zeta(x)\rangle \right] \\ &= \max_{\zeta \in \Xi} \|\mu_\zeta(\mathbb{P}) - \mu_\zeta(\mathbb{Q})\|_q \quad \text{(Dual)} \end{aligned} \tag{2.3}$$

The Dual equation in 2.3 requires Hölder's inequality. For $\|\omega\|_p \leq 1$, we have $\|\cdot\|_p$ to be the ℓ_p norm. Let $p,\ q \in [1, \infty]$, such that $\frac{1}{p} + \frac{1}{q} = 1$. Then,

$$|\langle x, y\rangle| \leq \|x\|_p \|y\|_q \tag{2.4}$$

We get the loss function from metric:

$$\mathscr{L}_\mu(\omega, \zeta, \theta) = \langle \omega, \mathbb{E}_{x\sim\mathbb{P}_r}\Phi_\zeta(x) - \mathbb{E}_{z\sim p_z}\Phi_\zeta(g_\theta(z))\rangle$$

For optimizing the model, we solve the mini-max problem:

$$\min_{g_\theta} \max_{\zeta \in \Xi} \max_{\omega, \|\omega\|_p \leq 1} \mathscr{L}_\mu(\omega, \zeta, \theta)$$

In practice, we have the non-parametric data sample set instead of the parametric data distribution. Therefore, we use the sample mean instead of the theoretical mean.

$$\hat{\mathscr{L}}(\omega, \zeta) = \left\langle \omega, \frac{1}{m}\sum_{i=1}^m \Phi_\zeta(x_i) - \frac{1}{m}\sum_{i=1}^m \Phi_\zeta(g_\theta(z_i)) \right\rangle \tag{2.5}$$

Then we apply the Stochastic Gradient Descent (SGD):

$$(\omega, \zeta) \leftarrow (\omega, \zeta) + \eta(\nabla_\omega \hat{\mathscr{L}}(\omega, \zeta), \nabla_\zeta \hat{\mathscr{L}}(\omega, \zeta))$$

where η is the learning rate. We will briefly talk about the choice of learning rate later in training part.

2.1.3 Geometric GAN

For the efficiency of training process, we usually separate the training dataset into several mini-batches in practice. The minibatch size n is much smaller than the data dimension d. This is called as high-dimensional low-sample size (HDLSS) problem. The mean difference (MD) classifier is a popular method to improve the performance of discriminator (Lim and Ye 2017). The Mean Difference method is using the normal vector ω to be the the hyperplane.

Instead of fitting the Statistical Distribution of data, Lim and Ye 2017 works on apply Support Vector Machine to the optimization problem. We will discuss about the Support Vector Machine in its own section. Here we mainly discuss about the application of Support Vector Machine on GAN.

Geometric Interpretation of Mean Feature Matching GAN

To get a geometric meaningful ω, we apply the a property of Cauchy-Schwarz inequality:

Theorem 4 (Cauchy-Schwarz (Steele 2004)) $|\langle u, v\rangle| \leq \|u\|\|v\|$

Corollary 5 *If* $|\langle u, v\rangle| = \|u\|\|v\|$, *then* $u = \dfrac{\langle u, v\rangle}{\|v\|^2} v$

In our case, let ω to be u, and $\frac{1}{m}\sum_{i=1}^{m}\Phi_\zeta(x_i) - \frac{1}{m}\sum_{i=1}^{m}\Phi_\zeta(g_\theta(z_i))$ to be v. We have $\|\omega\| \leq 1$. Then 2.5 has maximization to be when $\|\omega\| = 1$. Therefore, $\max \| \langle u, v\rangle \| = \|v\|$ Then we apply the corollary to get

$$u = \frac{\|v\|}{\|v\|^2} v = \frac{v}{\|v\|^{1/2}} \tag{2.6}$$

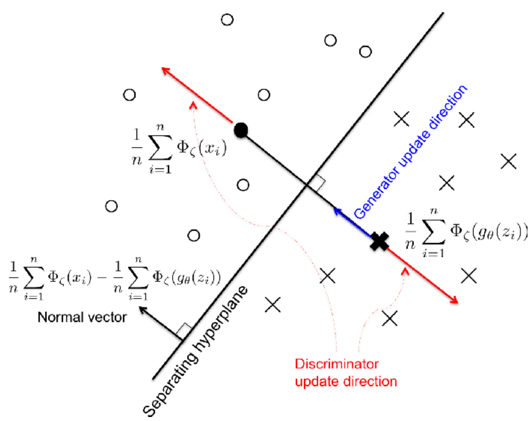
Take the value of u and v, we get

$$\omega^* = c\sum_{i=1}^{m}(\Phi_\zeta(x_i) - \Phi_\zeta(g_\theta(z_i)))/m$$

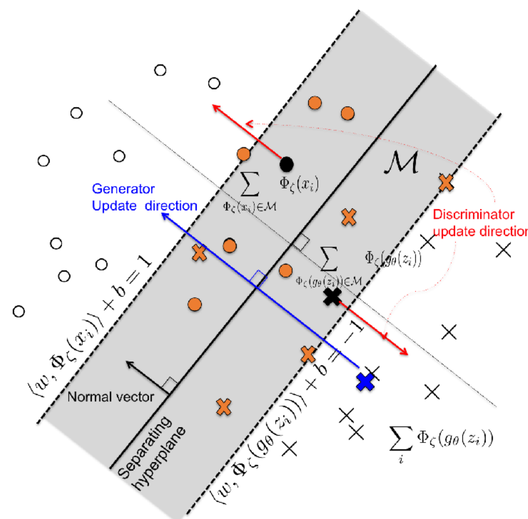
where $c = \| \sum_{i=1}^{m}(\Phi_\zeta(x_i) - \Phi_\zeta(g_\theta(z_i)))/m\|^{-1/2}$ (Lim and Ye 2017). From the ω^*, we can get the new formula of updating parameters:

$$
\begin{aligned}
\zeta &\leftarrow \zeta + \eta \sum_{i=1}^{m} \langle \omega^*, \nabla_\zeta \Phi_\zeta(x_i) - \nabla_\zeta \Phi_\zeta(g(z_i)) \rangle /m \\
\theta &\leftarrow \theta + \eta \sum_{i=1}^{m} \langle \omega^*, \nabla_\theta \Phi_\zeta(g(z_i)) \rangle /m
\end{aligned}
\tag{2.7}
$$

The derivation by Lim and Ye 2017 is different from the original McGAN (Mroueh *et al.* 2017).



(a) The Geometric Interpretation of McGAN



(b) The Geometric Interpretation of Geometric GAN

Figure 2.2: From both plots, we can see that both methods creates a linear classifier to discriminate the dataset. (Lim and Ye 2017)

From the vector ω^* and the image 2.2a, we can see that McGAN creates a hyperplane to separate the real data between generated data. When we compare the geometric interpretation between mean difference and SVM, the SVM creates a more robust hyperplane as a linear classifier. From the SVM classifier we have:

$$
\omega^{SVM} = \sum_{i=1}^{m} \alpha_i \phi_\zeta(x_i) - \sum_{i=1}^{m} r_i \phi_\zeta(g_\theta(z_i)) \tag{2.8}
$$

Here α_i, and r_i follows the definition in subsection SVM to be the Lagrangian multipliers. There also exists a region $\mathcal{M}$ between the margin boundaries to be

$$\mathcal{M} = \{\phi \in \mathbb{R}^M | \ \left|(\omega^{SVM} \cdot \phi) + b\right| \leq 1\}. \tag{2.9}$$

The points inside the region $\mathcal{M}$ are the Support Vectors. According to Lim and Ye 2017, the with ω^{SVM} and b, Lagrangian multiplier 2.20 can be:

$$\begin{aligned}
\mathscr{L}_\theta(\omega, b, \zeta) &= \frac{1}{m}\sum_{i \in I_S}(\omega^{SVM} \cdot \phi_\zeta(g_\theta(z_i))) - \frac{1}{m}\sum_{i \in I_T}(\omega^{SVM} \cdot \phi_\zeta(x_i)) + C && (2.10)\\
&= \frac{1}{m}\sum_{i=1}^{m}(\omega^{SVM} \cdot (s_i\phi_\zeta(g_\theta(z_i)) - t_i\phi_\zeta(x_i))) + C && (2.11)\\
& && (2.12)
\end{aligned}$$

Here (t_i, s_i) are geometric scaling factors defined by

$$t_i = \begin{cases} 1, & \phi_\zeta(x_i) \in \mathcal{M} \\ 0, & \text{otherwise} \end{cases}, \qquad s_i = \begin{cases} 1, & \phi_\zeta(g_\theta(z_i)) \in \mathcal{M} \\ 0 & \text{otherwise} \end{cases} \tag{2.13}$$

Since loss function of SVM only related with Support Vectors. From the new loss function, we will have different Gradient Descent function to update the parameters:

$$\zeta \leftarrow \zeta + \eta \sum_{i=1}^{m}(\omega^{SVM} \cdot (t_i\nabla_\zeta\phi_\zeta(x_i)) - s_i\nabla_\zeta\phi_\zeta(g_\theta(z_i)))))/m \tag{2.14}$$

$$\theta \leftarrow \theta + \eta \sum_{i=1}^{m}(\omega^{SVM} \cdot \nabla_\theta\phi_\eta(g_\theta(z_i))))/m \tag{2.15}$$

The gradient Descents of updating the Geometric GAN are similar with the ones for Mean Matching GAN. The main difference would be the existence of support vectors. We can also view Geometric GAN to be a special Mean Matching GAN where only the support vectors to be the elements considered in.

Loss Functions

In Lim and Ye 2017's work, they included and compared different loss functions from different kinds of Generative Adversial Nets. Here we mainly look into the algorithm of Hinge Loss. Hinge Loss is the loss function for Support Vector Machine

Algorithm 2 Hinge Loss. Input: $\phi(x_i)$ or $\phi(g(z_i))$. We will call them as ϕ_i in the algorithm.

Require: margin≥ 0
  for $i = 1, \cdots, m$ **do**
    pre-loss$_i$ = margin $- y_i \cdot \phi_i$
    loss$_i$ = $\mathbb{1}_{\text{pre-loss}_i \geq 0} \times$ pre-loss$_i$
  end for
  Loss $= \sum_{i=1}^{m} \text{loss}_i / m$

2.2 One Class Support Vector Machine

2.2.1 Support Vector Machine

The support Vector Method is a function estimating method with labelled data proposed by Scholkopf (1999). In the setting of SVM, we have

$$(x_1, y_1), \cdots (x_m, y_m).$$

y_i is the label (or measurement) of the vector x_i, and $y = f(x)$ holds. It is generally applied in pattern recognition and linear operations problems. The function $f(\cdot)$ may not be a linear function. To find out the relationship between x and y, we need to map the vector x from a n-dimensional space X into a unknown high dimensional (could be infinite) space Z, $y = \beta \cdot z$, where β is the linear parameter for the function above.

We start from x and y has linear relationship. With the training dataset, we have

$$(x_1, y_1), \cdots, (x_m, y_m),\ x \in X \subset \mathbb{R}^n,\ y \in \{1, -1\}$$

Assume the data is separable by the hyperplane

$$(w \cdot x) + b = 0$$

where there exists a vector w and constant b fulfills

$$\begin{aligned} &(w \cdot x_i) + b > 1, && \text{if } y_i = 1 \\ &(w \cdot x_j) + b < -1, && \text{if } y_j = -1 \end{aligned}$$

Vectors x_i such that fulfills $y_i(w \cdot x_i + b) = 1$ are *Support Vectors*. Then we combine two inequalities and their conditions, we can get the constraints

$$y_i(w \cdot x_i + b) \geq 1 \tag{2.16}$$

The optimal hyperplane minimizes $\Phi = w \cdot w$ Cortes and Vapnik 1995. From this, we can construct a Lagrangian

$$L(w, b, A) = \frac{1}{2} w \cdot w - \sum_{i=1}^{m} \alpha_i [y_i(w \cdot x_i + b) - 1] \tag{2.17}$$

where $A^T = (\alpha_1, \cdots, \alpha_m)$ is the non-negative Lagrange multiplier corresponding to the constraint (2.4). From gradients, we have

$$\begin{aligned} \frac{\partial L}{\partial w}|_{w=w_0} &= (w_0 - \sum_{i=1}^{m} \alpha_i y_i x_i) = 0 \\ \frac{\partial L}{\partial b}|_{b=b_0} &= \sum_{\alpha_i} y_i \alpha_i = 0 \end{aligned} \tag{2.18}$$

Therefore, we have

$$w_0 = \sum_{i=1}^{m} \alpha_i y_i x_i$$

for the convex optimization.

Soft Margin

The hypothesis of data is separable by a hyperplane cannot be fulfilled for all dataset. Therefore, Cortes and Vapnik 1995 developed Soft margin hyperplane to allow a small number of error. We introduce the error to be $\xi_i \geq 0,\ i = 1, \cdots m$ We want to minimize the functional

$$\Phi(\xi) = \sum_{i=1}^{m} \xi^{\sigma} \tag{2.19}$$

and the constraints becomes

$$\begin{aligned} y_i(w \cdot x_i + b) &\geq 1 - \xi_i & i &= 1, \cdots, m \\ \xi_i &\geq 0 & i &= 1, \cdots, m \end{aligned}$$

Then the Lagrange function becomes

$$L(w, \xi, b, A, R) = \frac{1}{2} w \cdot w + C(\sum_{i=1}^{m} \xi_i)^k - \sum_{i=1}^{m} \alpha_i [y_i (w \cdot_i + b) - 1 + \xi_i] - \sum_{i=1}^{m} r_i \xi_i \quad (2.20)$$

Here $R^T = (r_1, \cdots, r_m)$ is also the non-negative Lagrange multiplier for the constraints. To update the parameter w and b, we have:

$$\begin{aligned}
\frac{\partial L}{\partial w}|_{w=w_0} &= (w_0 - \sum_{i=1}^{m} \alpha_i y_i x_i) = 0 \\
\frac{\partial L}{\partial b}|_{b=b_0} &= \sum_{\alpha_i} y_i \alpha_i = 0 \\
\frac{\partial L}{\partial \xi_i}|_{\xi_i=\xi_i^0} &= kC(\sum_{i=1}^{m} \xi_i^0)^{k-1} - \alpha_i - r_i
\end{aligned} \quad (2.21)$$

After derive the equation, we have

$$w_0 = \sum_{i=1}^{m} \alpha_i y_i x_i$$

Kernel Trick

In the paragraphs above, we mainly discussed about the linear separation. When the dataset is not linear separable, we need to transform our dataset into a new vector (Cortes and Vapnik 1995). Define the transformation function ϕ:

$$\phi : \mathbb{R}^n \to \mathbb{R}^N, x \mapsto \phi(x)$$

where

$$\phi(x_i) = (\phi_1(x_i), \phi_2(x_i), \cdots, \phi_N(x_i)),\ i = 1, \cdots m$$

Then the decision function is:

$$f(x) = sgn(w \cdot \phi(x) + b) \quad (2.22)$$

This makes the vector w to be

$$w = \sum_{i=1}^{m} y_i \alpha_i \phi(x_i) \quad (2.23)$$

Combine the decision function with the vector w from the Lagrange, we get:

$$f(x) = sgn(\phi(x) \cdot w + b) = \sum_{i=1}^{m} y_i \alpha_i \phi(x) \phi(x_i) + b$$

By constructing a dot-product in Hilbert space (Anderson and Bahadur 1962), we have:

$$\phi(u) \cdot \phi(v) \equiv K(u, v).$$

For any symmetric $K(u, v) \in \mathcal{L}_2$, we can expend it into infinite summation according to Coutant and Hilbert 1953:

$$K(u, v) = \sum_{i=1}^{\infty} \lambda_i \phi_i(u) \cdot \phi_i(v)$$

That makes the decision function to be

$$f(x) = sgn(\sum_{i=1}^{m} y_i \alpha_i K(x, x_i)) \tag{2.24}$$

where $\alpha_i \geq 0$ and $\alpha > 0$ only for support vectors.

2.2.2 One-Class SVM

The algorithm of One Class SVM is simliar with the classic two class SVM. Instead of labelled data, we have unlabeled training data (Schölkopf *et al.* 2001)

$$x_1, \cdots, x_m \in X \subset \mathbb{R}^n$$

The algorithm of One-Class SVM is capturing the data points from one class inliers in small region and the outliers in the other area. It follows a optimization problem:

$$\begin{aligned} \min_{w \in \Omega, \Xi \in \mathbb{R}^m, b \in \mathbb{R}} \quad & \frac{1}{2} w \cdot w + \frac{1}{\nu m} \sum_{i=1}^{m} \xi_i + b \\ \text{subject to} \quad & (w \cdot \Phi(x_i)) + b \geq -\xi_i, \quad \xi_i \geq 0 \end{aligned} \tag{2.25}$$

The OCSVM does not contain the information about the labels. Therefore, the subjective function does not provide us a symmetric inequality like the original SVM.

$$
\begin{aligned}
y_i(w \cdot \Phi(x_i)) + b \geq -\xi_i & \qquad \text{subjective function of original GAN} \\
y_i(w \cdot \Phi(x_i)) \leq \xi_i + b & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (2.26) \\
|w \cdot \Phi(x_i)| \leq \xi_i + b & \qquad\qquad\qquad\qquad y_i \text{ is either } 1 \text{ or } -1 \qquad (2.27)
\end{aligned}
$$

The subjective function of original SVM is symmetric, while for the OCSVM, we do not have such property. Therefore, the final soft boundary of OCSVM is not symmetric.

From the optimization problem, we have the decision function to be:

$$f(x) = sgn(w \cdot \phi(x) + b)$$

The Lagrangian to be:

$$L(w, b, \xi, A, R) = \frac{1}{2} w \cdot w + \frac{1}{\nu l} \sum_{i=1}^{m} \xi_i + b - \sum_{i=1}^{m} \alpha_i (w \cdot \phi(x_i) + b + \xi) - \sum_{i=1}^{m} r_i \xi_i \quad (2.28)$$

where A, R are the multipliers similar to the classic SVM.

The derivatives are also similar with the two-classes SVM:

$$
\begin{aligned}
w &= \sum_{i=1}^{m} \alpha_i \phi(x_i) \\
\alpha_i &= \frac{1}{\nu l} - r_i
\end{aligned}
\quad (2.29)
$$

The decision function can also be:

$$f(x) = sgn(\sum_{i=1}^{m} \alpha_i K(x_i, x) + b)$$

Convex Optimization and Gradient Descent

In classic Support Vector Machine algorithm, we use convex optimization to find the closed-form solution. This includes applying the Karush–Kuhn–Tucker condition. In this situation, we have restricted selections about the kernels. While in deep neural network, we generally lose the convexity of functions. In this situation, we cannot use the Duality and functions from Schölkopf *et al.* 2001. Instead, we use the gradient descent to find a local optimal. Therefore, in this section, we did not discuss about the dual problem and the choices of the kernel $K(\cdot, \cdot)$.

2.3 Gap in Research

For all the loss functions we mentioned above, we only classify samples belong to either Real or Generated Distribution. In classification problems, our data is collected from limited subspaces. Therefore, all the samples belongs to one of those classes. However, in generative model, we are generating data from the whole space. The samples might belong to neither class. We need to take a linear separation after the space transformation. It is possible that the data is well separated but the generation is not well defined. Therefore, there are more than two classes (Real v.s. Generated) of data. We conclude it as Correctly Classified Real Data, Correctly Classified Generated Data, and Incorrectly Classified Data. If our discriminator is too powerful and override the generator, we will have the generator not updating. Compared with labelled classification, the unsupervised learning One Class SVM restricts a small area that contains most data. If we apply unsupervised discriminator, we may have a more robust model compared with the supervised discriminator. However, we cannot update unsupervised discriminator with the labels we have. In this case, we need to adjust the algorithm to update the networks while keep the property of unsupervised learning.

2.4 Conclusions

In this Chapter, we briefly reviewed the classic and other popular algorithms of Generative Adversial Nets. Most of them based on the probability properties. While the Geometric-GAN by Lim and Ye 2017 discovered a method with the geometric interpretation. From the geometric methods, we can then apply more different classifiers including the Support Vector Machine.

The second half of the paper, we went through the classic Support Vector Machine and the One-Class Support Vector Machine. Both methods belonged to the convex optimization. The classic Support Vector Machine is a supervised method to classify labelled data. The One-Class SVM is an unsupervised method generally for outlier detection and novelty detection. The algorithm of both methods are similar and both including kernel methods to apply to complicated dataset. While in our case, we use Gradient Descent to solve the SVM problems. Then we no longer need the restrictions from the Convex Optimizations. Our kernels can be flexible as the deep networks could achieve. However, the Gradient Descents does not guarantee the global minimum/ maximum as convex optimization. That brings more difficulty in tuning the model.

Chapter 3

Methods and Procedure

In this paper, we propose a discriminator using the One-Class Support Vector Machine. We will discuss the methods and algorithm in this chapter.

3.1 One-Class SVM Hyperplane

From the background information above, we have briefly understood the algorithm of One-Class SVM (OCSVM) and its usage. For each OCSVM classifier, we have the output to be inliers and outliers. If we interpret into numbers, that would be $+1$ (inliers) and -1 (outliers). When we have a OCSVM classifier for GAN, naturally we want to see the real data to be classified as inliers ($+1$) while the generated data to be outliers (-1). We also notice that the OCSVM is an unsupervised learning model. That implies that we cannot update this model with the labels we have. Therefore, we need to adjust the loss functions from the OCSVM and allow us to penalize the data not be in the classified as the same class they should be.

As we discussed from 2.4, there exists the three different classes in the Dataset in practice, and we want to separate all these classes simultaneously. Since we do not have the labels of these Classes, we simply point out which elements of the dataset fulfill our requirements. To achieve this, we place two OCSVM hyperplanes in the discriminator: one with the real data to be the inlier, another one with the generated data to be the inliers. Those data are classified to be the inliers only for the OCSVM if their class is correctly classified. The others are the 'Incorrectly' Classified data.

Table 3.1: The Labels from Different Class of Data to Different Classifier

		Classes of Data	
		Real Data	Generated Data
Classifier	Real Classifier	+1	-1
	Generated Classifier	-1	+1

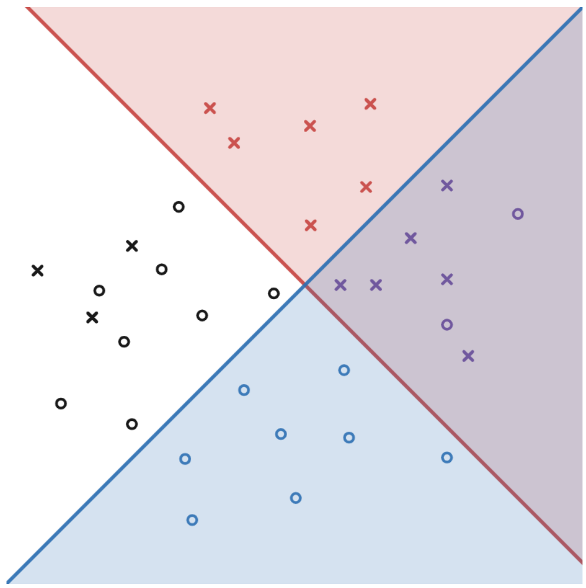


Figure 3.1: The implication of our discriminator. We have red and blue separating lines. The red and blue area are the area of inliers for each classifier. Then the red and blue dots are correctly classified by both classifiers, while the purple and black dots are the inliers for both separators and outliers for both separators respectively.

3.1 is a simulation about how the real and generated data can be divided by two linear classifiers. The plot did not show the optimized result, while there are still misclassified data points. From the plot 3.1 and table 3.1, we can see that we can divide the data into four groups. We penalize on the data misclassified on one or both classifiers. Generally we use the convolutional network to project the images to a higher dimensional space $\phi(\cdot)$. Project all the data to the same space $\phi(\cdot)$ to avoid overfitting problem.

From 3.2, we show that the Discriminator is same for both OCSVM classifiers.

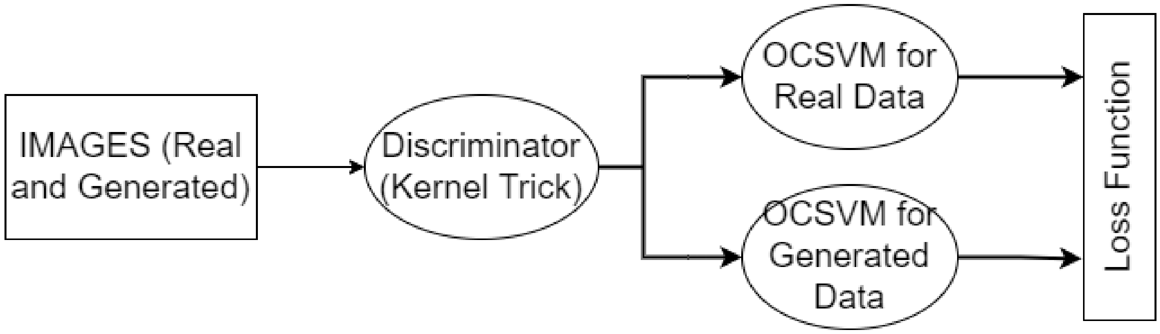


Figure 3.2: Discriminator of OCSVM-GAN.

After the automatic kernel trick, we apply the data into both classifiers and get two loss values.

3.1.1 Updating the Unsupervised Network

There are two parts in our SVM function. One is the loss value, the distance between the target label and the trained label. Another one is the weight. That indicates the support vectors. In each OCSVM function, we apply a linear function to downsize the data to a real value.

We need to notice that for OCSVM, the optimization problem 2.25 subject to

$$\omega \cdot \phi(x) \geq \rho - \xi_i, \ \xi \geq 0. \tag{3.1}$$

Combine this with the decision function

$$f(x) = sgn(\omega \cdot \phi(x) - \rho)$$

we can get that all the inliers and 'not-too-off' outliers are support vectors as shown in 3.3.

Before we define the loss function of the total loss, we need define the loss function for thereal classifier and the generated classifier separately. Similar with the Geometric GAN, we first define the region between the margin boundaries $\mathcal{M}$:

$$\mathcal{M}_{real} = \{\phi_\zeta \in \mathbb{R}^M | w_{real} \cdot \phi_\zeta + b_{real} \leq 1\} \tag{3.2}$$

$$\mathcal{M}_{gen} = \{\phi_\zeta \in \mathbb{R}^M | w_{gen} \cdot \phi_\zeta + b_{gen} \leq 1\} \qquad \text{the parameters are different} \tag{3.3}$$

For each $\mathcal{M}_{\text{type}}$, we have two indicator functions

$$t_i^{\text{type}} = \begin{cases} 1, & \phi_\zeta(x_i) \in \mathcal{M} \\ 0, & \text{otherwise} \end{cases}, \qquad s_i^{\text{type}} = \begin{cases} 1, & \phi_\zeta(g_\theta(z_i)) \in \mathcal{M} \\ 0 & \text{otherwise} \end{cases} \tag{3.4}$$

Then the Lagrangian multiplier for real classifier would be

$$\begin{aligned} \mathscr{L}_\theta^{\text{real}}(\omega_{\text{real}}, b, \zeta) &= \frac{1}{m}\sum_{i\in I_S}(\omega^{\text{real}} \cdot \phi_\zeta(g_\theta(z_i))) - \frac{1}{m}\sum_{i\in I_T}(\omega^{\text{real}} \cdot \phi_\zeta(x_i)) + C && (3.5)\\ &= \frac{1}{m}\sum_{i=1}^{m}(\omega^{\text{real}} \cdot \big(s_i^{\text{real}}\phi_\zeta(g_\theta(z_i)) - t_i^{\text{type}}\phi_\zeta(x_i)\big)) + C && (3.6) \end{aligned}$$

Similarly, the Lagrangian multiplier for generated classifier would be

$$\begin{aligned} \mathscr{L}_\theta^{\text{gen}}(\omega_{\text{gen}}, b_{\text{gen}}, \zeta) &= \frac{1}{m}\sum_{i\in I_T}(\omega^{\text{gen}} \cdot \phi_\zeta(x_i)) - \frac{1}{m}\sum_{i\in I_S}(\omega^{\text{gen}} \cdot \phi_\zeta(g_\theta(z_i))) + C && (3.7)\\ &= \frac{1}{m}\sum_{i=1}^{m}(\omega^{\text{gen}} \cdot (t_i^{\text{gen}}\phi_\zeta(x_i)) - s_i^{\text{gen}}\phi_\zeta(g_\theta(z_i))) + C && (3.8) \end{aligned}$$

Combine both Lagrangian multiplier together, we get

$$\begin{aligned} \mathscr{L}_\theta(w_{\text{real}}, w_{\text{gen}}, \zeta) = \mathscr{L}_\theta^{\text{real}} & \quad (\omega_{\text{real}}, b_{\text{real}}, \zeta) + \mathscr{L}_\theta^{\text{gen}}(\omega_{\text{gen}}, b_{\text{gen}}, \zeta) && (3.9)\\ = \frac{1}{m} & \quad \sum_{i=1}^{m}(\omega^{\text{real}} \cdot \big(s_i^{\text{real}}\phi_\zeta(g_\theta(z_i)) - t_i^{\text{type}}\phi_\zeta(x_i)\big) && (3.10)\\ & \quad + \omega^{\text{gen}} \cdot (t_i^{\text{gen}}\phi_\zeta(x_i)) - s_i^{\text{gen}}\phi_\zeta(g_\theta(z_i))) + C && (3.11) \end{aligned}$$

Since we have the labels for each data point in our case, we can update out network through the Gradient Descents optimization while penalize on part of the outliers. We split the loss functions into two parts: the loss value l, and the weight w. Here we use score function to differ from the Loss function above. It is still the non-negative number; not the negative number from the classic definition of Score function.

3.2 Experiment & Tuning

We test the performance of our algorithm on the famous MNIST dataset. We compare the result from Geometric GAN (Lim and Ye 2017). We use the same generator and

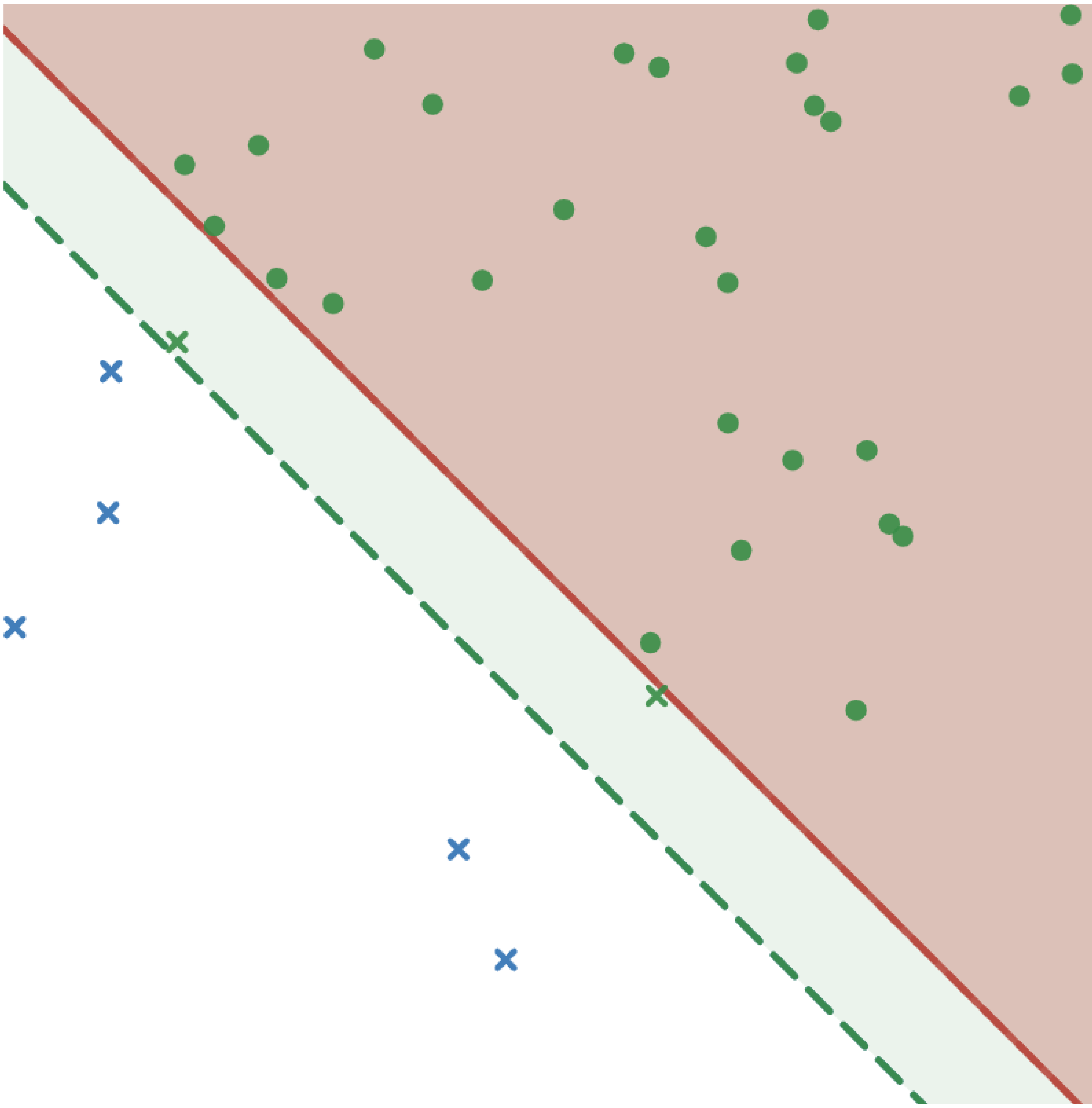


Figure 3.3: The Indication Plot of OCSVM: The red straight line is the separating hyperplane, and the red area is the area of inliers. All the round points are the inliers in our sample data. The green dashed line is the soft margin with respect to the the given margin distance. All the blue points will not affect the final result of the classification while all the green points are the support vectors.

discriminator network for both methods. In general, we have a better result than result from Geometric GAN (Lim and Ye 2017).

'Training on Real Testing on Synthetic' (TRTS), 'Training on Synthetic, Testing on Real' (TSTR) are popular methods to evaluate the GAN models. While in our cases, we do not have the labelling process and the labelled classification cannot

Algorithm 3 The Loss Function of OCSVM

Target : label of the Class of OCSVM
Class: the input class
margin: the soft margin for OCSVM, defined by the user
input: x n-dimensional data
$\hat{y} \leftarrow \beta x$ $\qquad \triangleright$ linear regression $\mathbb{R}^n \rightarrow \mathbb{R}$
if margin $- \hat{y} \times Target \geq 0$ **then**
    LOSS VALUE $\leftarrow |\hat{y} \times Class - Target|$
end if
return LOSS VALUE

Figure 3.4: The example images from MNIST Dataset

present the quality of the images. It is useful in more complex contexts. We also considered about the Inception Score Barratt and Sharma 2018, while the empirical calculating of Inception Score falls into the distribution of our model. Therefore, we will compare the images directly to show the quality of generated images and evaluate the performance.

3.2.1 Network

We did not use a deep network for training due to the computation reason. Figure3.5 presents our model network. For both generator and discriminator, there are five parametric layers. They are symmetric to each other. The fully connected layer is the linear regression operation. It is mainly working on the SVM and OCSVM classification as we discussed above.

3.3 Results and Discussion

Tuning Generative Adversarial Network is more complicated than the classification-only networks. The engineering part involves many potential risks and situations. The most typical one is the mode collapsing.

If we train the model with large number of epochs, we may see the mode collapse. The generated images will be the identical images. Since we want to generate a variety of images, it is not desired situation. In GAN network, we generally use **ADAM Optimizer** to solve the problem (Goodfellow 2016). The two hyper-parameters can adjust the decay rate of the learning rate. In our model, the results seems collapsed (the non-stable part from the loss plot 3.6b before it getting better result.

Combine these images and plots, we can see that both models converges. Since the Geometric GAN does not create an effective image, we can say that our model has better performance in the same network architecture.

We need to notice that from when comparing different GAN models, the objective functions are different. In Geometric GAN and OCSVMGAN, we have different

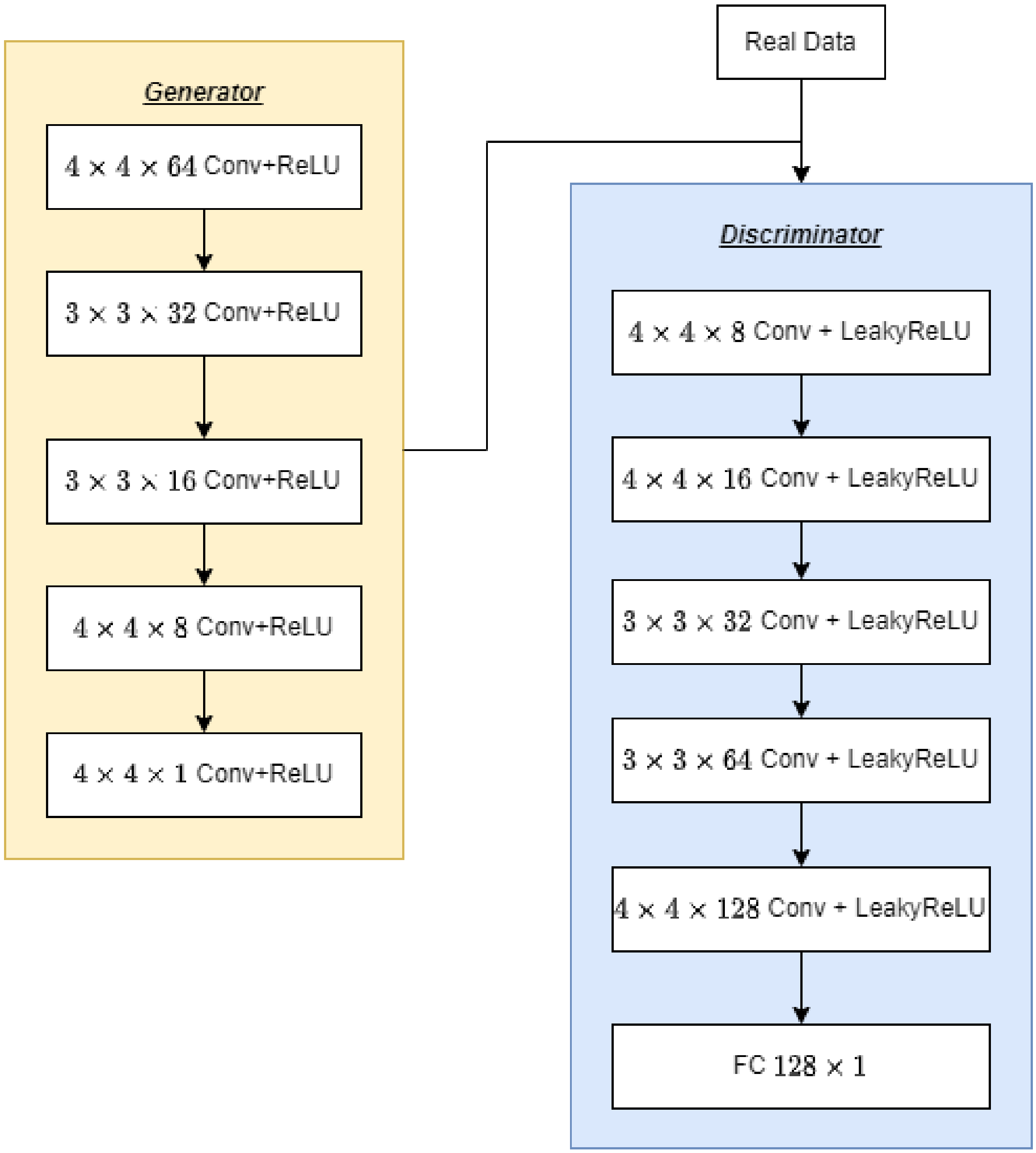
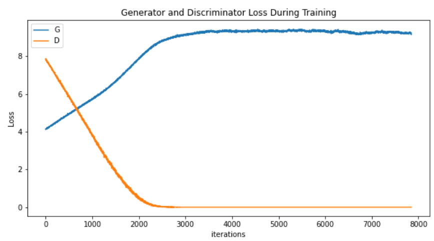
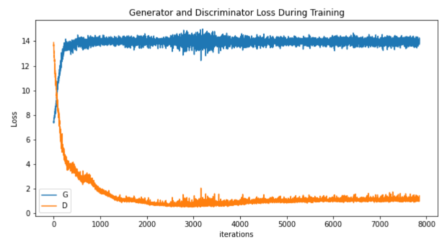


Figure 3.5: The Architecture of Network in Our Experiments

objective functions if we have different margins. Therefore, the loss value cannot be qualitative measure of performance of GAN. The loss value plot is a effective method to observe the convergence of the networks.

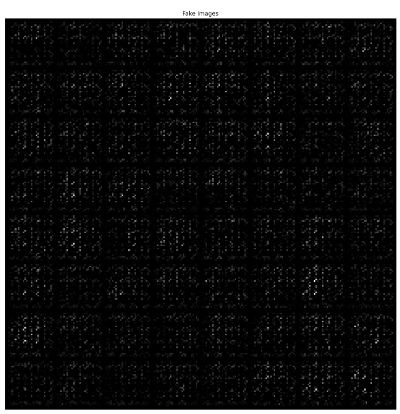


(a) The loss Value of Geometric GAN

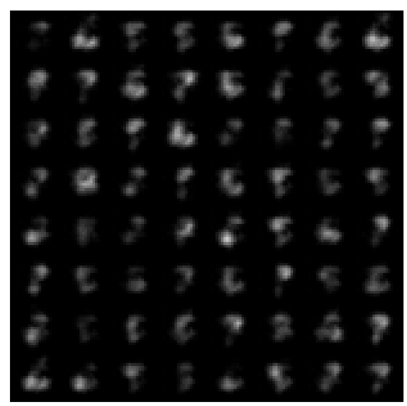


(b) The loss value of Our Model

Figure 3.6: As shown in the plots, out model converges faster than the Geometric GAN.



(a) The The Generated images of Geometric GAN



(b) The generated images of Our Model

Figure 3.7: In this plot, we can see that the Geometric does not any useful information. While our model has generated digits.

3.4 Conclusions

We use OCSVM to take the advantage of its soft-margin and support vectors from the classic SVM. The OCSVM is used to be an unsupervised method, but we can still apply the labels to update the model efficiently.

We also applied two separators in discriminators. In the traditional Generative models, we only classes the images to be real/generated classes. The generator can create images have close patterns to the real data but nothing close to the real images. We defined three classes for the whole dataset. The new one is the 'Incorrectly Classified Data', with respect to two 'Correctly Classified Data' (Real and Generated). In this way we have the similar idea of separating multiple classes Vapnik 1991. However, we do not use three separating parameters since the exact location for 'Wrong Data' is not our main attention. Our method can eliminate the underfitting problem in traditional generative models.

In the experiment part, we can see that our model has outperformed than the Geometric GAN in the setting with same networks. From what result we have, I believe with further fine tuning, we can see better and clear images from our model. The MNIST dataset is popular in Computer Vision and GAN, while there are many datasets that require deeper network and longer training time. We believe our model can provide a better solution in those fields.

Chapter 4

Conclusions, Recommendations, & Future Work

4.1 Conclusions

To conclude, we proposed a new method of Generative Adversarial Network. In this network we followed the geometric definition from Lim and Ye 2017. While in our algorithm, we separate the whole dataset into three parts instead of two part in other algorithms. These three parts are: the correctly classified real data, the correctly classified generated data, and the incorrectly classified data. We want to find the space of the correctly classified data respectively. In the same time, we try to eliminate the space of the incorrectly data. Our network uses the One Class Support Vector Machine, and it is an un/semi-supervised method. We can still assign the labels to the classifiers and operate classification. We compared our classifier with Lim and Ye 2017's classifier in the same network, and our network performs better than theirs. It shows that our model works in the same setting.

4.2 Future Work

For the model, we can operate more precise tuning method to see whether we can have a better output. Then we can experiment on more complex datasets. Two SVMs increase the number of hyperparameters we can tune. Now we are using the same margin for both Support Vector Machine, while we can discuss about the effect of different margins.

For the algorithm, our model uses two OCSVMs to be the classifiers for the real and generated data . We can also try to discriminate the real and generated data by using two classic SVM. In this way, we will have different margins and support vectors.

Bibliography

[1] J. H. Lim and J. C. Ye, "Geometric gan," *arXiv preprint arXiv:1705.02894*, 2017.

[2] I. Goodfellow, "Nips 2016 tutorial: Generative adversarial networks," *arXiv preprint arXiv:1701.00160*, 2016.

[3] I. Goodfellow *et al.*, "Generative adversarial nets," *Advances in neural information processing systems*, vol. 27, 2014.

[4] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein generative adversarial networks," in *International conference on machine learning*, PMLR, 2017, pp. 214–223.

[5] Y. Mroueh, T. Sercu, and V. Goel, "Mcgan: Mean and covariance feature matching gan," in *International conference on machine learning*, PMLR, 2017, pp. 2527–2535.

[6] J. M. Steele, *The Cauchy-Schwarz master class: an introduction to the art of mathematical inequalities.* Cambridge University Press, 2004.

[7] B. Scholkopf, C. J. C. Burges, and A. J. Smola, *Advances in Kernel Methods : Support Vector Learning.* MIT Press, 1999, ISBN: 9780262194167. [Online]. Available: https://login.ezproxy.library.ualberta.ca/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=e000xna&AN=421&site=ehost-live&scope=site.

[8] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[9] T. W. Anderson and R. R. Bahadur, "Classification into two multivariate normal distributions with different covariance matrices," *The annals of mathematical statistics*, pp. 420–431, 1962.

[10] R Coutant and D. Hilbert, *Methods of mathematical physics*, 1953.

[11] B. Schölkopf, J. C. Platt, J. Shawe-Taylor, A. J. Smola, and R. C. Williamson, "Estimating the support of a high-dimensional distribution," *Neural computation*, vol. 13, no. 7, pp. 1443–1471, 2001.

[12] S. Barratt and R. Sharma, "A note on the inception score," *arXiv preprint arXiv:1801.01973*, 2018.

[13] V. Vapnik, "Principles of risk minimization for learning theory," *Advances in neural information processing systems*, vol. 4, 1991.

[14] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," *arXiv preprint arXiv:1511.06434*, 2015.

[15] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International conference on machine learning*, PMLR, 2015, pp. 448–456.

[16] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Icml*, 2010.

Appendix A: Related Networks

A.1 DC-GAN

Radford *et al.* 2015 proposed a new algorithm of GAN using Convolutional Neural Network (CNN). They named network as Deep Convolutional GAN (DCGAN). Before that, the attempts of GAN using CNN were not successful. In this paper, Radford *et al.* adopted some changes to CNN architectures to achieve a satisfying result. The first change is that they no longer apply spatial pooling functions. Instead, they applied the strided convolutions for down/upsampling. The second is to eliminate the fully connected layers, especially for the middle layers. The fully connected layers will decrease the convergence speed, but it can also increase the model stability.

They also identified that application of the batchnorm (Ioffe and Szegedy 2015) directly would cause sample oscillation and model instability. Thus, they use ReLU activation (Nair and Hinton 2010) instead.

By combining all these changes, Radford *et al.* proposed the new structure using CNN to generate the images as show in image A.1. Lim and Ye 2017 also applied the architecture of DCGAN. We tested our algorithm utilizing OCSVM in both DCGAN and the architecture we adjusted from DCGAN.

Lim and Ye claimed that they tested their algorithm within the network of DC-GAN. We also applied the network of DC-GAN to both our model and Geometric GAN. Since the MNIST dataset is 28×28, we upscaled the dataset into 96×96. Then we apply the Sigmoid before we calculated the loss value. However, we did not get good results from either network. This might because the tuning parameters do not match the network or the dataset. It requires future works to see the results.

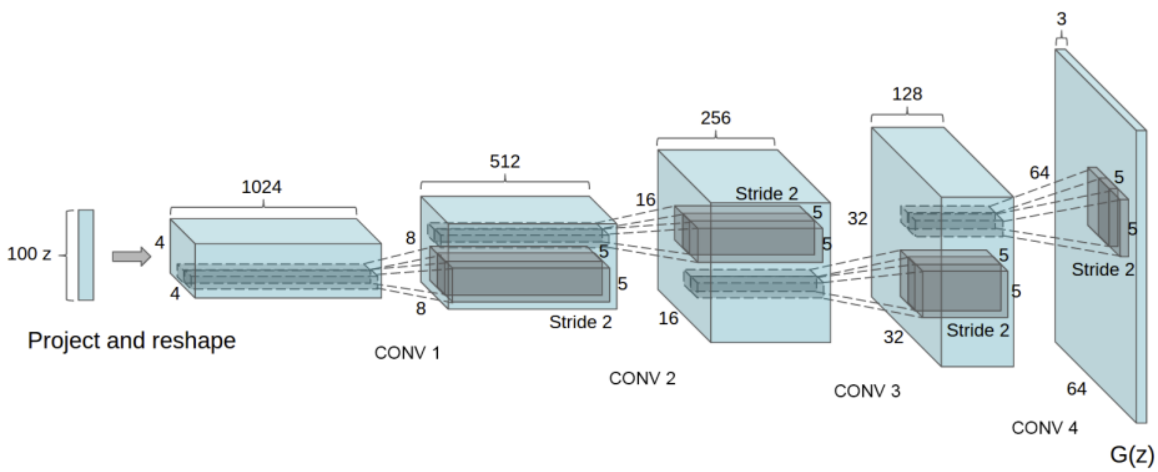


Figure A.1: The Generator Architecture of DCGAN (Radford *et al.* 2015)

Appendix B: Experiment Codes

B.1 OCSVM GAN

```
# -*- coding: utf-8 -*-
"""OCSVMGAN MNIST (One NetD).ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1fX7kw0LZv3qduq2N4JFxJfddh1bhdWyP
"""

# -*- coding: utf-8 -*-
"""
DCGAN Tutorial
==============

**Author**: `Nathan Inkawhich <https://github.com/inkawhich>`__

"""

from __future__ import print_function
#%matplotlib inline
import argparse
import os
import random
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.backends.cudnn as cudnn
import torch.optim as optim
import torch.utils.data
import torchvision.datasets as dset
```

```
import torchvision.transforms as transforms
import torchvision.utils as vutils
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML

# Set random seed for reproducibility
#manualSeed = 999
manualSeed = random.randint(1, 10000) # use if you want new results
print("Random Seed: ", manualSeed)
random.seed(manualSeed)
torch.manual_seed(manualSeed)


################
# Inputs
# ------
#
#  Lets define some inputs for the run:
#
# -  **dataroot** - the path to the root of the dataset folder. We will
#    talk more about the dataset in the next section
# -  **workers** - the number of worker threads for loading the data with
#    the DataLoader
# -  **batch_size** - the batch size used in training. The DCGAN paper
#    uses a batch size of 128
# -  **image_size** - the spatial size of the images used for training.
#    This implementation defaults to 64x64. If another size is desired,
#    the structures of D and G must be changed. See
#    `here <https://github.com/pytorch/examples/issues/70>`__ for more
#    details
# -  **nc** - number of color channels in the input images. For color
#    images this is 3
# -  **nz** - length of latent vector
# -  **ngf** - relates to the depth of feature maps carried through the
```

```
#    generator
# -  **ndf** - sets the depth of feature maps propagated through the
#    discriminator
# -  **num_epochs** - number of training epochs to run. Training for
#    longer will probably lead to better results but will also take much
#    longer
# -  **lr** - learning rate for training. As described in the DCGAN paper,
#    this number should be 0.0002
# -  **beta1** - beta1 hyperparameter for Adam optimizers. As described in
#    paper, this number should be 0.5
# -  **ngpu** - number of GPUs available. If this is 0, code will run in
#    CPU mode. If this number is greater than 0 it will run on that number
#    of GPUs
#


# Number of workers for dataloader
workers = 2

# Batch size during training
batch_size = 128

# Spatial size of training images. All images will be resized to this
#   size using a transformer.
image_size = 28

# Number of channels in the training images. For color images this is 3
nc = 1

# Size of z latent vector (i.e. size of generator input)
nz = 100

# Size of feature maps in generator
ngf = 8

# Size of feature maps in discriminator
```

```
ndf = 8

# Number of training epochs
num_epochs = 50

# Learning rate for optimizers
lr = 0.00001

# Beta1 hyperparam for Adam optimizers
beta1 = 0.9

# Number of GPUs available. Use 0 for CPU mode.
ngpu = 1


############
# Data
# ----

import pandas as pd

MNIST = pd.read_csv('/content/sample_data/
   mnist_train_small.csv', header=None)
images = MNIST.iloc[0:, 1:]
images = np.asarray(images)
images = images.astype('float').reshape(-1,1,28,28)
images_tensor = torch.from_numpy(images)
images=images_tensor.float()



dataloader = torch.utils.data.DataLoader(images,
   batch_size=batch_size,
                                          shuffle=True,
                                            num_workers=
                                            workers)

# Decide which device we want to run on
device = torch.device("cuda:0" if (torch.cuda.is_available
   () and ngpu > 0) else "cpu")

# Plot some training images
real_batch = next(iter(dataloader))
plt.figure(figsize=(8,8))
plt.axis("off")
```

```
plt.title("Training Images")
plt.imshow(np.transpose(vutils.make_grid(real_batch.to(device)[:64], padding=2, normalize=True).cpu(),(1,2,0)))

print(real_batch.shape)



##############
# Implementation
# --------------
#
# With our input parameters set and the dataset prepared, we can now get
# into the implementation. We will start with the weight initialization
# strategy, then talk about the generator, discriminator, loss functions,
# and training loop in detail.
#
# Weight Initialization
# ~~~~~~~~~~~~~~~~~~~~~
#
# From the DCGAN paper, the authors specify that all model weights shall
# be randomly initialized from a Normal distribution with mean=0,
# stdev=0.02. The ``weights_init`` function takes an initialized model as
# input and reinitializes all convolutional, convolutional-transpose, and
# batch normalization layers to meet this criteria. This function is
# applied to the models immediately after initialization.
#

# custom weights initialization called on netG and netD
def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)
```

```


######################################################################
# Generator
# ~~~~~~~~~
#
# The generator, :math:`G`, is designed to map the latent space vector
# (:math:`z`) to data-space. Since our data are images, converting
# :math:`z` to data-space means ultimately creating a RGB image with the
# same size as the training images (i.e.  3x64x64 ). In practice, this is
# accomplished through a series of strided two dimensional convolutional
# transpose layers, each paired with a 2d batch norm layer and a relu
# activation. The output of the generator is fed through a tanh function
# to return it to the input data range of :math:`[-1,1]`. It is worth
# noting the existence of the batch norm functions after the
# conv-transpose layers, as this is a critical contribution of the DCGAN
# paper. These layers help with the flow of gradients during training. An
# image of the generator from the DCGAN paper is shown below.
#
# .. figure:: /_static/img/dcgan_generator.png
#    :alt: dcgan_generator
#
# Notice, the how the inputs we set in the input section (*nz*, *ngf*, and
# *nc*) influence the generator architecture in code. *nz* is the length
# of the z input vector, *ngf* relates to the size of the feature maps
# that are propagated through the generator, and *nc* is the number of
# channels in the output image (set to 3 for RGB images). Below is the
```

```
# code for the generator.
#

# Generator Code

class Generator(nn.Module):
    def __init__(self, ngpu):
        super(Generator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is Z, going into a convolution
            nn.ConvTranspose2d( nz, ngf * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            # state size. (ngf*8) x 4 x 4
            nn.ConvTranspose2d(ngf * 8, ngf * 4, 3, 1, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            # state size. (ngf*4) x 8 x 8
            nn.ConvTranspose2d( ngf * 4, ngf * 2, 3, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            # state size. (ngf*2) x 16 x 16
            nn.ConvTranspose2d( ngf * 2, ngf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),
            # state size. (ngf) x 32 x 32
            nn.ConvTranspose2d( ngf, nc, 4, 2, 1, bias=False),
            nn.ReLU(True)
            # state size. (nc) x 28 x 28
        )

    def forward(self, input):
        return self.main(input)


###############################################################################
# Now, we can instantiate the generator and apply the ``weights_init``
```

```
# function. Check out the printed model to see how the generator object is
# structured.
#

# Create the generator
netG = Generator(ngpu).to(device)

# Handle multi-gpu if desired
if (device.type == 'cuda') and (ngpu > 1):
    netG = nn.DataParallel(netG, list(range(ngpu)))

# Apply the weights_init function to randomly initialize all weights
#  to mean=0, stdev=0.02.
netG.apply(weights_init)

# Print the model
print(netG)


class Discriminator(nn.Module):
    def __init__(self, ngpu):
        super(Discriminator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is (nc) x 28 x 28
            nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf) x 32 x 32
            nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*2) x 16 x 16
            nn.Conv2d(ndf * 2, ndf * 4, 3, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),
            ##############Adding ectra layers between these two
            nn.Conv2d(ndf * 4, ndf * 4, 3, 1, 1, bias=False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),
            #nn.Dropout(0.2),
```

```
            ###########################################################
            # state size. (ndf*4) x 8 x 8
            nn.Conv2d(ndf * 4, ndf * 8, 3, 1, 1, bias=False),
            nn.BatchNorm2d(ndf * 8),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*8) x 4 x 4
            nn.Conv2d(ndf * 8, ndf * 16 , 4 , 1, 0, bias=False),
            #nn.Sigmoid()
        )

    def forward(self, input):
        return self.main(input)

def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)

# Create the Discriminator
netD = Discriminator(ngpu).to(device)

# Handle multi-gpu if desired
if (device.type == 'cuda') and (ngpu > 1):
    netD = nn.DataParallel(netD, list(range(ngpu)))

# Apply the weights_init function to randomly initialize all weights
#  to mean=0, stdev=0.2.
netD.apply(weights_init)

# Print the model

from torch.autograd import Variable
class OCSVM_R(nn.Module):

  def __init__(self, margin=3.5, size_average=True, sign=1.0):
    super(OCSVM_R, self).__init__()
    self.ngpu = ngpu
```

```
    self.sign = sign
    self.margin = margin
    self.size_average = size_average


    self.main = nn.Linear(ndf * 16, 1)

  def forward(self, input, target):
    input = input.view((b_size, -1))

    #
    #input = self.main(input)

    #input = torch.sigmoid(self.main(input))
    input = self.main(input)
    #print(input.shape)
    input = input.view(-1)
    #print(input.shape)


    assert input.dim() == target.dim()

    for i in range(input.dim()):

        assert input.size(i) == target.size(i)
    #output now is the loss function (giving out weight of decision function)
    #First OCSVM sign 1
    #
    #output = torch.mul(target, input)

    #\phi(x_i)+\xi >= 0
    output =  self.margin - torch.mul(input, 1 *target) #(There is no longer direction)
    #output = self.margin - input          #output is the loss for OCSVM
    y = input

    #pull the weight of too fitted inliners into 1
    y_loss = torch.mul (y, 1) -target * 1
    #y_loss[torch.gt(y_loss, 1)] = 1
    y_loss = torch.abs(y_loss)

    #

```

```

    #
    if 'cuda' in input.data.type():
        mask = torch.cuda.FloatTensor(input.size()).zero_()
    else:
        mask = torch.FloatTensor(input.size()).zero_()
    mask = Variable(mask)
    mask[torch.gt(output, 0.0)] = 1.0
    #mask[torch.lt(output, -1.0)] = -1.0
    #output[torch.gt(output, 1.0)] = 1.0

    #output = (output > mask).float() - torch.sigmoid(output - mask).detach() + torch.sigmoid(output - mask)


    #
    output = torch.mul(output, mask)
    #output = mask

    # apply sign
    #print("output", output)
    #print("y", y)
    output = torch.mul(output, y_loss)


    # size average
    if self.size_average:
        output = torch.mul(output, 1.0 / input.nelement())

    # sum
    output = output.sum()
    #print(output)
    # apply sign
    #output = torch.mul(output, self.sign)
    #output = torch.mul (output, 1) #sign for real OCSVM

    #output = output - self.sign * (1)
    return output
    #we use each single output to be weight y. Then we create a weighted classification distance

from torch.autograd import Variable
class OCSVM_G(nn.Module):
```

```
  def __init__(self, margin=3.5, size_average=True, sign=1.0):
    super(OCSVM_G, self).__init__()
    self.sign = sign
    self.ngpu = ngpu
    self.margin = margin
    self.size_average = size_average

    self.main = nn.Linear(ndf * 16, 1)



  def forward(self, input, target):

      input = input.view((b_size, -1))

      #input = self.main(input)

      input = self.main(input)##############should I change sigmoid into something else now it's (0,1) #(-infty, infty)


      #
      input = input.view(-1)

      #
      #assert input.dim() == target.dim()
      #for i in range(input.dim()):
          #assert input.size(i) == target.size(i)

      #Second OCSVM sign -1
      #
      #output = torch.mul(target, input)
      output =  self.margin - torch.mul(input, -1* target) #(There is no longer direction) (CANNOT see the direction)
      #output = self.margin - input
      y = input

      y_loss = torch.mul (y, -1) - target * -1
      #y_loss[torch.gt(y_loss, 1)] = 1
      y_loss = torch.abs(y_loss)
      #y_loss = torch.abs(torch.mul (y, -1) - target * -1)
```

```
         #
        if 'cuda' in input.data.type():
            mask = torch.cuda.FloatTensor(input.size()).zero_()
        else:
            mask = torch.FloatTensor(input.size()).zero_()
        mask = Variable(mask)
        mask[torch.gt(output, 0.0)] = 1.0
        #mask[torch.lt(output, -1.0)] = -1.0

        #output = (output > mask).float() - torch.sigmoid(output - mask).detach() + torch.sigmoid(output - mask)

        #
        output = torch.mul(output, mask)
        #output = mask

        # apply sign
        output = torch.mul(output, y_loss)


        # size average
        if self.size_average:
          output = torch.mul(output, 1.0 / input.nelement())

        # sum
        output = output.sum()
        #print(output)

        # apply sign
        #output = torch.mul(output, self.sign)
        #output = torch.mul (output, -1) #sign for real OCSVM

        #output = output - self.sign * (-1)
        return output

# Initialize BCELoss function
criterion_R = OCSVM_R(size_average = True)
criterion_G = OCSVM_G(size_average = True)
```

```
fixed_noise = torch.randn(64, nz, 1, 1, device=device)



# Establish convention for real and fake labels during
   training
real_label = 1.
fake_label = -1.

# Setup Adam optimizers for ONLY D
optimizerD = optim.Adam(netD.parameters(), lr=lr, betas=(
   beta1, 0.9))

# Setup Adam optimizer for G
optimizerG = optim.Adam(netG.parameters(), lr=lr, betas=(
   beta1, 0.9))

# Commented out IPython magic to ensure Python
   compatibility.
from locale import strcoll
# Training Loop

# Lists to keep track of progress
img_list = []
G_losses = []
D_losses = []
iters = 0

print("Starting Training Loop...")
# For each epoch
for epoch in range(num_epochs):
    # For each batch in the dataloader
    for i, data in enumerate(dataloader , 0):
        #print(real.shape)
        ############################
        # (1) Update D network: maximize log(D(x)) + log(1
            - D(G(z)))
        ###########################
        ## Train with all-real batch
        #label = label.data.resize_(batch_size).fill_(
           real_label)
        for p in netD.parameters(): # reset requires_grad
            p.requires_grad = True # they are set to False
                below in netG update
```

```
        for p in netG.parameters():
            p.requires_grad = False # to avoid computation

        netD.zero_grad()
        # Format batch
        real_cpu = data.to(device)
        #print(data.shape)
        b_size = real_cpu.size(0)
        #print(real_cpu.size(0))
        label = torch.full((b_size,), real_label, dtype=torch.float, device=device)
        # Forward pass real batch through D
        output = netD(real_cpu)
        #print(label.shape)
        # Calculate loss on all-real batch
        errD_rr = criterion_R(output, label)
        errD_rg = criterion_G(output, label)
        # Calculate gradients for D in backward pass
        errD_rr.backward(retain_graph =True)
        errD_rg.backward(retain_graph =True)
        D_x = output.mean().item()

        ## Train with all-fake batch
        noise = torch.randn(b_size, nz, 1, 1, device=device)
        fake = netG(noise)
        label.fill_(fake_label)
        #print(real.shape)
        # Classify all fake batch with D
        output = netD(fake.detach()).view(-1)
        # Calculate D's loss on the all-fake batch
        errD_gr = criterion_R(output, label)
        errD_gg = criterion_G(output, label)
        # Calculate the gradients for this batch, accumulated (summed) with previous gradients
        errD_gr.backward(retain_graph=True)
        errD_gg.backward(retain_graph=True)
        D_G_z1 = output.mean().item()
        # Compute error of D as sum over the fake and the real batches
        errD = errD_rr + errD_rg + errD_gr + errD_gg
        errD.backward()
        # Update D
        optimizerD.step()

```

```
        ############################
        # (2) Update G network: maximize log(D(G(z)))
        ###########################
        for p in netD.parameters():
            p.requires_grad = False # to avoid computation
        for p in netG.parameters():
            p.requires_grad = True # reset requires_grad

        netG.zero_grad()
        label.fill_(real_label)  # fake labels are real for generator cost ########################
        # Since we just updated D, perform another forward pass of all-fake batch through D
        fake = netG(noise)
        output = netD(fake).view(-1)
        # Calculate G's loss based on this output
        errGR = criterion_R(output, label)
        errGG = criterion_G(output, label)
        errGR.backward(retain_graph=True)
        errGG.backward(retain_graph=True)
        errG =errGR+errGG
        # Calculate gradients for G
        errG.backward()
        D_G_z2 = output.mean().item()
        # Update G
        optimizerG.step()

        # Output training stats
        if i % 50 == 0:
            print('[%d/%d][%d/%d]\tLoss_D: %.4f\tLoss_G: %.4f\tD(x): %.4f\tD(G(z)): %.4f / %.4f'
#                   % (epoch, num_epochs, i, len(dataloader),
                      errD.item(), errG.item(), D_x, D_G_z1, D_G_z2))

        # Save Losses for plotting later
        G_losses.append(errG.item())
        D_losses.append(errD.item())

        # Check how the generator is doing by saving G's output on fixed_noise
        if (iters % 100 == 0) or ((epoch == num_epochs-1) and (i == len(dataloader)-1)):
```

```
            with torch.no_grad():
                fake = netG(fixed_noise).detach().cpu()
            img_list.append(vutils.make_grid(fake, padding=2, normalize=True))
            # Plot the fake images from the last epoch
            plt.subplot(1,2,2)
            plt.axis("off")
            #plt.title("Fake Images")
            plt.imshow(np.transpose(img_list[-1],(1,2,0)))
            plt.savefig("/content/drive/MyDrive/THESIS/TRAIN/"+ str(iters*epoch) + ".png")
            plt.show()

        iters += 1










##################
# Results
# -------
#
# Finally, lets check out how we did. Here, we will look at three
# different results. First, we will see how D and  G s losses changed
# during training. Second, we will visualize  G s  output on the fixed_noise
# batch for every epoch. And third, we will look at a batch of real data
# next to a batch of fake data from G.
#
# **Loss versus training iteration**
#
# Below is a plot of D &  G s  losses versus training iterations.
#

plt.figure(figsize=(10,5))
```

```
plt.title("Generator and Discriminator Loss During Training")
plt.plot(G_losses,label="G")
plt.plot(D_losses,label="D")
plt.xlabel("iterations")
plt.ylabel("Loss")
plt.legend()
plt.savefig("OCSMLOSS.png")
plt.show()



#############
# **Visualization of  G s  progression**
#
# Remember how we saved the generators output on the fixed_noise batch
# after every epoch of training. Now, we can visualize the training
# progression of G with an animation. Press the play button to start the
# animation.
#

#%%capture
fig = plt.figure(figsize=(8,8))
plt.axis("off")
ims = [[plt.imshow(np.transpose(i,(1,2,0)), animated=True)] for i in img_list]
ani = animation.ArtistAnimation(fig, ims, interval=1000, repeat_delay=1000, blit=True)

HTML(ani.to_jshtml())


##################
# **Real Images vs. Fake  Images**
#
# Finally, lets take a look at some real images and fake images side by
# side.
#

# Grab a batch of real images from the dataloader
real_batch = next(iter(dataloader))
```

```

# Plot the real images
plt.figure(figsize=(15,15))
plt.subplot(1,2,1)
plt.axis("off")
plt.title("Real Images")
plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(
    device)[:64], padding=5, normalize=True).cpu(),(1,2,0))
    )

# Plot the fake images from the last epoch
plt.subplot(1,2,2)
plt.axis("off")
plt.title("Fake Images")
plt.imshow(np.transpose(img_list[-1],(1,2,0)))
plt.show()
plt.savefig("OCSVMGEN.png")
```

Appendix C: Simulation Comparison

We tried to replicate the simulation experiment from Geometric GAN (Lim and Ye 2017). We created 25 Gaussian mixture model, and applied the same network as their experiment design. Both Generator and Discriminator are fully connected.

- **Discriminator**: FC(2, 128)-ReLU-FC(128, 128)-ReLU-FC(128, 128)-ReLU-FC(128,1)
- **Generator**: FC(4, 128)-BN-ReLU-FC(128, 128)-BN-ReLU-FC(128, 128)-BN-ReLU-FC(128, 2) (Lim and Ye 2017)

From the simulation result, we can see that the W-GAN significantly does not cover the whole area. The rest three have much better performance than the performance of W-GAN. When we look into the GAN, Geometric GAN, and OCSVMGAN, we find out that they are all trying to cover the most area of the real distribution. While the distribution of GAN and Geometric GAN look similar, the OCSVMGAN has least impact on the original point (the lightest blue point). Therefore, we have some evidence to say that the OCSVM GAN is better than the other networks in reducing mode collaspes.

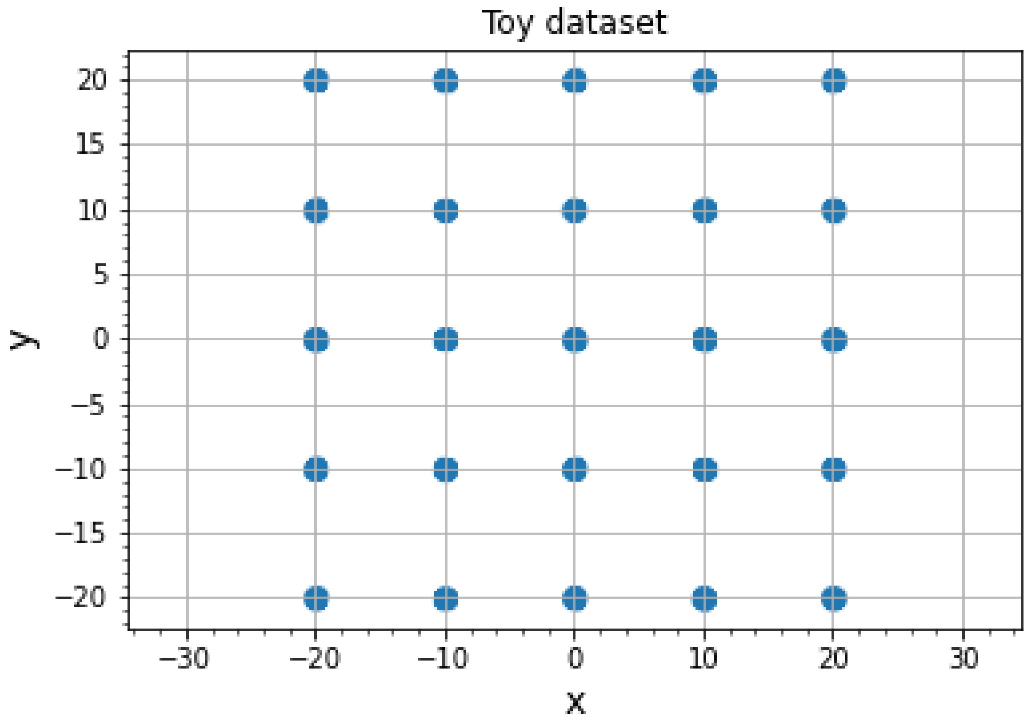


Figure C.1: The Distribution of Simulation Training Data

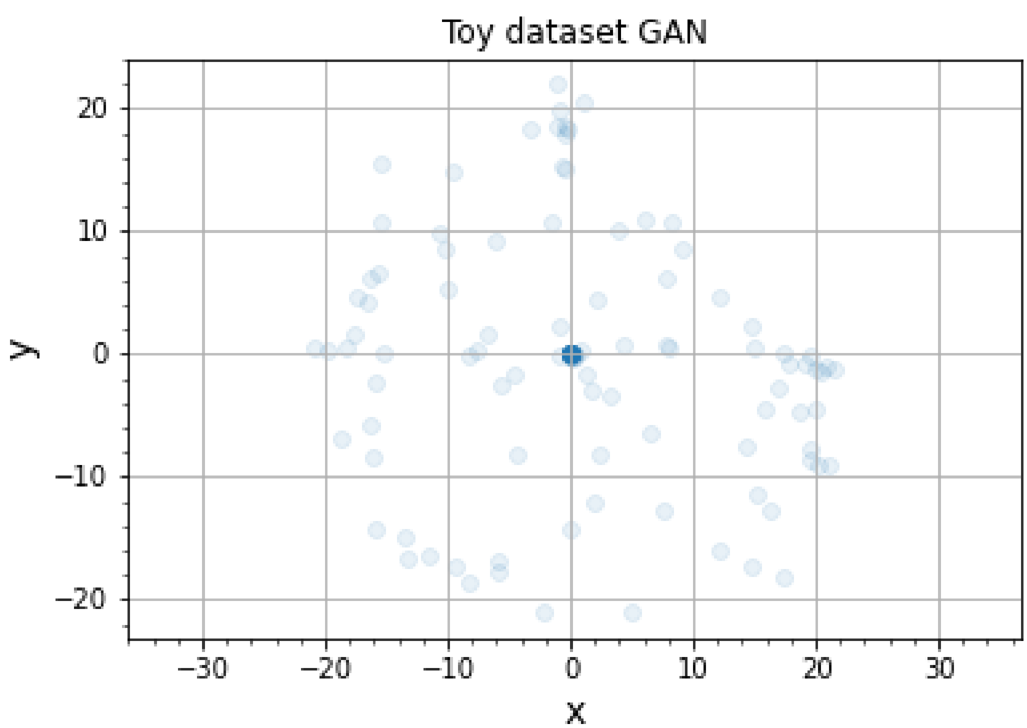


Figure C.2: The Distribution Generated Data from original GAN

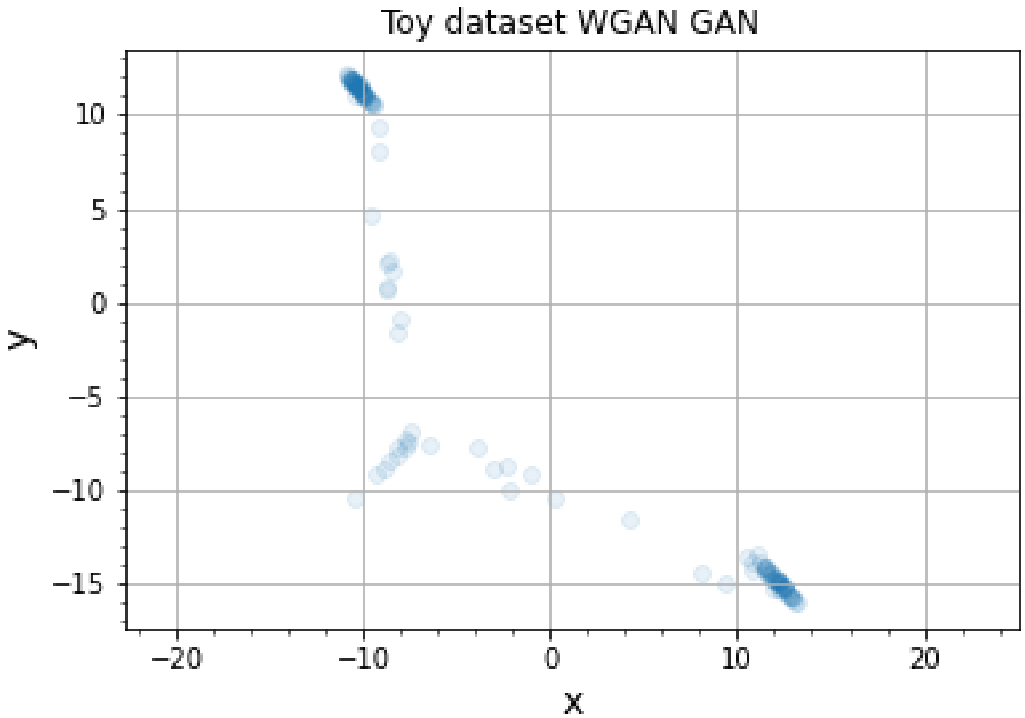


Figure C.3: The Distribution of Generated Data from W-GAN

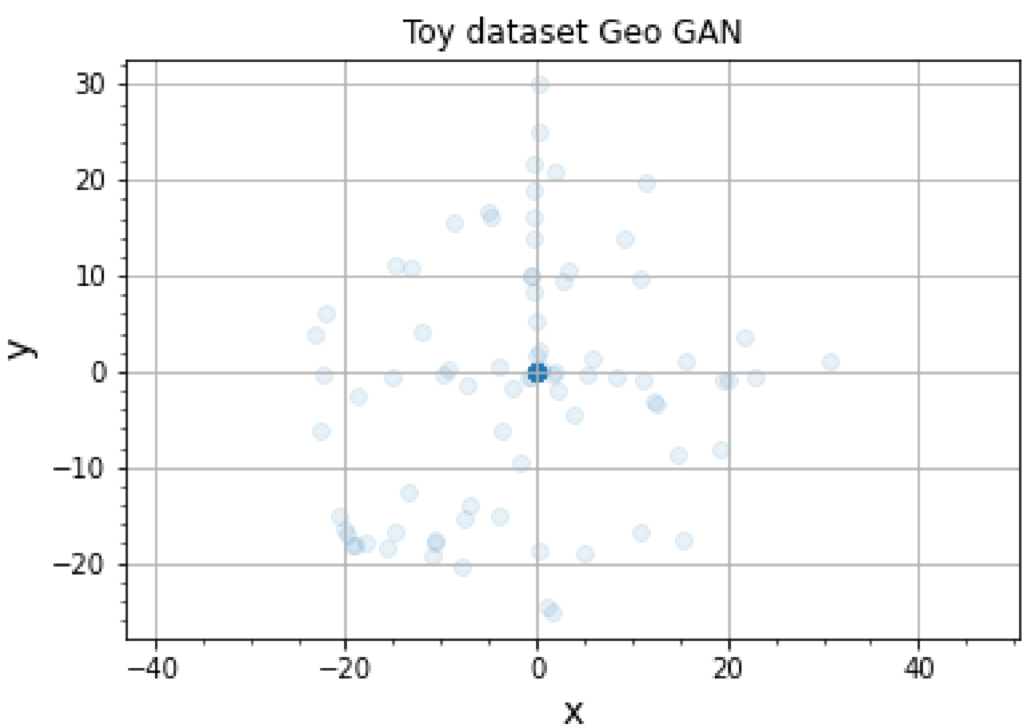


Figure C.4: The Distribution of Generated Data from Geometric GAN

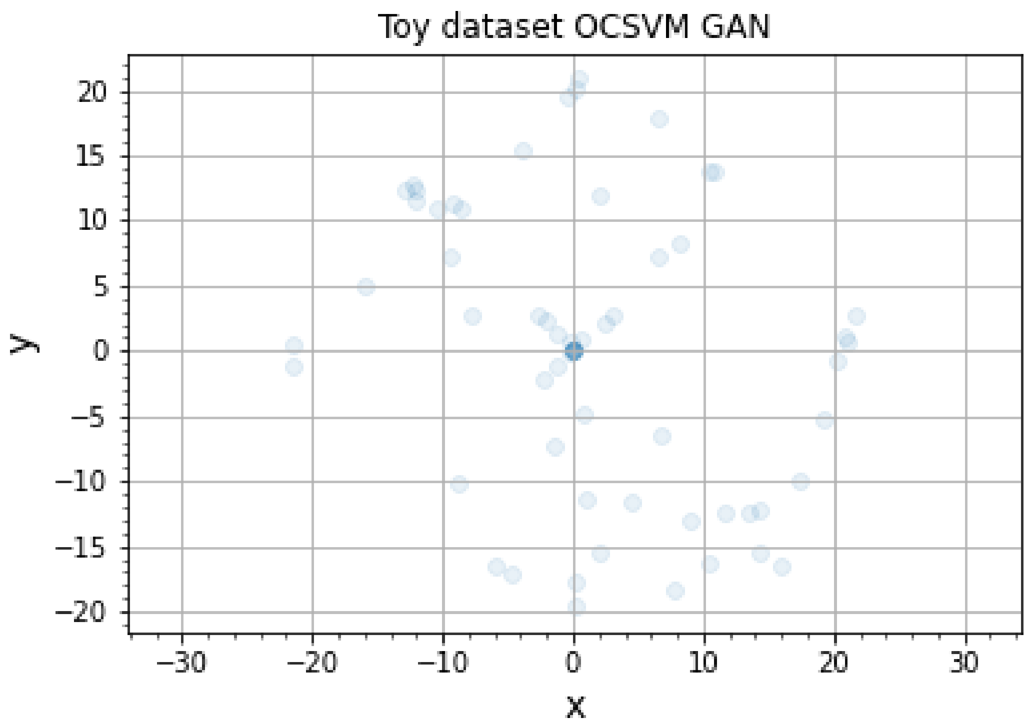


Figure C.5: The Distribution of Generated Data from OCSVMGAN