Generative Multi-Adversarial Neural Networks for Image Generation

Jim Jiayi Xu, Zhao Binglin, Xiufeng Zhao, Xiaoyin Yang, Liang Hou
Department of Electrical and Computer Engineering
University of California San Diego
La Jolla, 92093

{jjx002, bzhao, x6zhao, x4yang, l7hou}@eng.ucsd.edu

Abstract

Generative adversarial neural networks (GANs) are a class of machine learning architectures in which two neural networks are synchronously trained against each other in a feedback loop. The generator network learns to emulate distributions from a training set to generate realistic samples, while the discriminator network learns to classify samples as members of the training set or generated fakes. Ultimately, the generator is used to generate realistic samples indistinguishable from the dataset samples. We will test the performance of adding additional discriminator networks to the original GAN model to create generative multi-adversarial networks. We will evaluate how effective these alternative architectures are at generating imagery when compared to the original GAN. We will also test the performance of deep convolutional GANs (DCGANs) when we extend the architecture to a deep convolutional GMAN (DCGMAN). We make all direct comparisons between single adversarial and multiple adversarial models using the generative adversarial metric (GAM).

1 Introduction

1.1 Generative Adversarial Networks

A GAN is composed of two discrete component neural networks, the generator and the discriminator, each with competing objectives. The generator's goal is to emulate the training set; it uses function $G_\theta(z)$ with learned variables θ to transform its input random variable z to an output x that is indistinguishable from samples within that training set. The discriminator's goal is to classify which samples are real training samples and which samples were artificially generated by the generator; it uses function $D_\omega(x)$ with learned parameters ω to transform sample x to a binary output (1 if x is authentic, 0 if x is generated). During their synchronized training, Generator G and Discriminator D play a minimax game. D attempts to maximize $D_\omega(x)$, the probability of correctly classifying a real training image x and minimize $D_\omega(G_\theta(z))$, the probability of misclassifying the generator's outputs. G attempts to maximize $D_\omega(G_\theta(z))$. Therefore the overall GAN objective is expressed as follows:

$$\min_G \max_D V(D,G) = E_{x \sim P_{data}(x)}[logD(x)] + E_{z \sim p_z(z)}[log(1 - D(G(z))] \quad (1)$$

$$D_\omega(x) : x \rightarrow [0,1] \qquad G_\theta(z) : z \rightarrow x \quad (2)$$

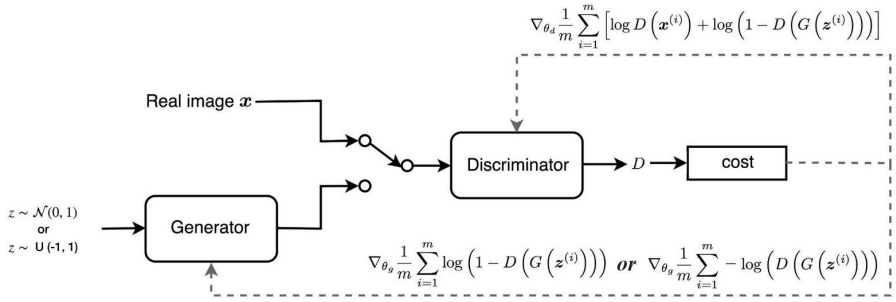


Figure 1: High-Level GAN Description. A GAN consists of two discrete neural networks that must be synchronously trained.

The distribution p_{data} is the underlying probability distribution of the training data, and the outputs of G implicitly define a probability distribution p_g. Through the two-player minimax game, G learns to approximate p_{data} with p_g. The theoretical solution to the game is when $p_g = p_{data}$, and D successfully classifies all real training samples but can only make random guesses about G's outputs. However, GANs often do not converge to this ideal solution, and the convergence of GANs is still under heavy research [2].

1.2 Deep Convolutional GAN (DCGAN) [5]

DCGAN is a popular and successful model that applies deep convolutional neural networks to GAN. In the generator, fractionally strided convolutions are used to up-sample the latent noise vector z to an appropriately sized image. In the discriminator, convolutional layers down-sample an image into a flat feature vector. The discriminator in a DCGAN uses no pooling networks. It learns its own downsampling through its convolutional layers. DCGANs are better suited for image generation than GANs because their learned convolutional filters capture high-level hierarchical features within the sample space. Also, the convolutional layers themselves impose constraints on the DCGAN architecture that makes them to more stable to train in image processing applications [5].

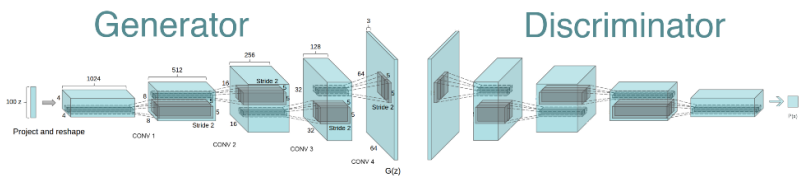


Figure 2: Deep Convolutional GAN Diagram

1.3 Motivation

Since Ian Goodfellow introduced GANs in *Generative Adversarial Nets* in 2014, people have been fascinated by them and their potential to learn any distribution of data from multiple domains: images, speech, video, etc. Leading experts in machine learning have deemed GANs as "the most interesting idea in the last 10 years in machine learning." However, GANs are notoriously difficult to train because one needs to synchronize its two discrete neural networks, such that one does not overpower the other. The networks need to be in constant balance with one another; otherwise, the GAN will fail to converge to an acceptable state. For example, if the discriminator is much more powerful than the generator, it will classify all of the generator's outputs as fake nearly 100% of the time and eliminate the gradient needed for both networks to train. Conversely, if the generator is much more powerful than the discriminator, it will continually produce the same samples to fool the weak discriminator. This state is called "mode collapse" and is characterized by nearly identical generator outputs despite variations in its random input.

Figure 3: Example of Mode Collapse on MNIST Data. The system converges and produces identical outputs despite changes in random input z.

Although there are many proposals that aim to circumvent these training difficulties, they reformulate the original GAN objective, such that generator G and discriminator D no longer play a zero-sum game. We explore the idea of adding multiple discriminator networks to a GAN to implement a generative multiple adversarial network that preserves the fundamental GAN objective yet eases the difficult training process[2].

1.4 Generative Multiple Adversarial Networks

Generative Multiple Adversarial Networks (GMAN) extends the base GAN model with additional discriminator networks. The generator is trained against the aggregated output of multiple discriminator networks, $F(V_1, V_2, \ldots V_N)$ where $V(D_i, G_i)$ denotes the value function of each D sub-network when paired with common generator G. Through proper selection of F, GMANs can achieve higher training performance by avoiding imbalances between the generator and discriminators. For example, if the generator frequently overpowers the overall discriminator network, one can easily create a more powerful discriminator with $F := max$. In this paper, we use $F := mean$ to aggregate the discriminator outputs, such that G will train against an ensemble of D networks. This requires 5 appropriate discriminator models. Since the feedback from the discriminators is averaged among the 5, any discriminator that is severely mismatched to the generator will degrade the performance of the entire GAN.

As we have previously stated, a noteworthy feature of GMANs is that they do not alter the primary objective function of the GAN network:

$$\min_G \max_D V(D,G) = E_{x \sim P_{data}(x)}[logD(x)] + E_{z \sim p_z(z)}[log(1 - D(G(z))]$$

Other proposals aim to reformulate this objective function into a more tractable form. For example, researchers in practice replace the term, $log(1 - D(G(z))$ with $-D(G(z))$. Although this has an added benefit of enhancing the performance of gradient descent techniques, this new formulation is no longer a zero-sum game. Although GMANs drastically changes the architectural design of GANs, they do not alter their fundamental objective of achieving a zero-sum game[2].

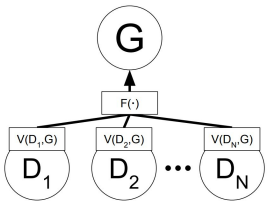


Figure 4: High-Level GMAN Description

2 Description of Method

2.1 Evaluation Metric [3]

Evaluating the performance of GANs is troublesome because there are few ways to quantitatively measure the outputs of the generators. Fortunately, the generative adversarial metric (GAM) allows us to directly compare the performance of one GAN against another. This metric requires two GAN models M_1 and M_2, each comprised of their constituent G and D networks.

$$M_1 = \{(G_1, D_1)\} \text{ and } M_2 = \{(G_2, D_2)\}$$

The two stages of using the GAM are the training phase and the testing phase[3]. During the training phase, you train each model by playing the GAN minimax game with its constituent networks. During the testing phase, you compare the performances of each GAN by swapping their D networks. In this testing phase, G_1 attempts to trick D_2 and G_2 attempts to trick D_1.

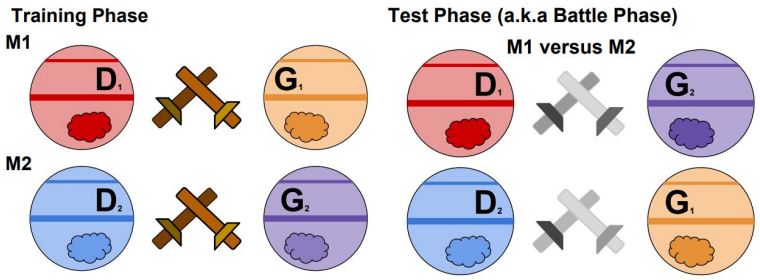


Figure 5: Training and Testing Phase Overview

After both stages of the GAM, you can evaluate their performances by examining the scores that the models obtained during the process.

	M_1	M_2
M_1	$D_1(G_1(\mathbf{z}))$, $D_1(\mathbf{x}_{train})$	$D_1(G_1(\mathbf{z}))$, $D_1(\mathbf{x}_{test})$
M_2	$D_2(G_2(\mathbf{z}))$, $D_2(\mathbf{x}_{test})$	$D_2(G_2(\mathbf{z}))$, $D_2(\mathbf{x}_{train})$

Figure 6: Table of Scores from the Generative Adversarial Metric The notation follows the formulation of the GAN objective function in section 1.1[1]

Intuitively, if G_1 can fool D_2 more than G_2 can fool D_1, then M_1 is objectively better than M_2 under the GAM. However, this metric has an additional restraint that M_1 and M_2 must be good GAN models. Therefore, for the GAM to produce a meaningful result, $D_1(x)$ and $D_2(x)$ must both be approximately 1. Therefore, the GAM produces a winner on the following criteria:

$$r_{test} \stackrel{\text{def}}{=} \frac{\epsilon\big(D_1(\mathbf{x}_{test})\big)}{\epsilon\big(D_2(\mathbf{x}_{test})\big)} \text{ and}$$

$$r_{samples} \stackrel{\text{def}}{=} \frac{\epsilon\big(D_1(G_1(\mathbf{z}))\big)}{\epsilon\big(D_2(G_2(\mathbf{z}))\big)},$$

$$\text{winner} = \begin{cases} \text{M1} & \text{if } r_{sample} < 1 \text{ and } r_{test} \simeq 1 \\ \text{M2} & \text{if } r_{sample} > 1 \text{ and } r_{test} \simeq 1 \\ \text{Tie} & \text{otherwise} \end{cases}$$

Figure 5: Scoring metric for the Generative Adversarial Metric [3]

We use this GAM to compare the performances of our GANs with their improved counterparts. In summary, it swaps the discriminator networks, such that each generator plays its adversarial minimax game against the opposing GAN's discriminator.

3 Implementation Details

To implement models we plan to use 2 datasets with different input size and image class. For each of these datasets, we create and compare the outputs from 4 GAN models: modified GAN, GMAN, DCGAN, and DCGMAN (utilizing both enhancements). We create GANs with fully connected layers to use as our control variable. They contain several convolutional layers, but they have only fully connected layers near the output. We create DCGANs with only convolutional layers and implement them with the architecture parameters and guidelines outlined in [5]. We created the GMANs by emulating the experiments in [2], using 5 discriminator networks with the mean function aggregator. They are created using convolutional layers only near the inputs and only fully connected layers near the outputs. Finally, we create the DCGMANs, which utilize both enhancements. We compare each of these augmented GANs with the control GANs with the GAM mentioned in Section 2.1. We do not use data augmentation. But for GAN and GMAN, we use soft labels to train the model. In addition, we sample from a gaussian distribution in latent space. Except for DCGAN and DCGMAN trained on MNIST which use an inputs training image size of 28x28, all other models are adapted for an input training image size of 32x32.

4 Experimental Settings

4.1 Datasets

We test our GANs by using them to generate imagery from the MNIST and CIFAR-10 datasets. MNIST dataset is an elementary dataset for computer vision, which consists of images of ten digits from 0 to 9. MNIST has a training set of 60,000 examples, and a test set of 10,000 examples. CIFAR-10 dataset consists of colorful images from 10 class: airplane, automobile, dog, cat, bird, frog, deer, horse, ship, truck. For each class, CIFAR-10 has 6000 images.

4.2 Training Parameters

For DCGAN and DCGMAN, we used mini-batch sizes of 100 on both the MNIST and CIFAR-10 data. For GAN and GMAN, we used mini-batch sizes of 64. All models were trained for 20 epoches using the Adam optimizer in place of mini-batch stochastic gradient descent. We use LeakyRelu with a slope of 0.2. We used 0.5 momentum and 0.0001 learning rates. DCGAN has 4 convolutional layers in and the regular GAN has 3 convolutional layers and 1 fully connected layers. We made sure to use batch normalization layers. These settings apply to both discriminator and generator models.

4.3 Hardware

We worked on both the UCSD server pod sessions and Google CoLab to develop all code in PyTorch. We used the GPU resources(Nvidia GTX 1080 Ti at school server, and Nvidia Tesla K80 at Google CoLab) allocated to us through these development platforms. Runtime typically lasts between 10-20 minutes for 20 epochs for each of the GANs and datasets.

5 Results

5.1 MNIST Outputs

The GAN models are trained for 20 epochs on MNIST data and the losses of the networks are plotted below in the following figures. As expected, the DCGAN has more stable training behavior than the GAN, with fewer spikes in generator and discriminator loss. Also, the GMAN and DCGMAN have even more stable behavior than their single discriminator counterparts. The outputs of the generators are also shown below in the following figure. We include these figures for qualitative comparison only. We use a more objective and quantitative comparison with our generative adversarial metric (GAM) in the following section.

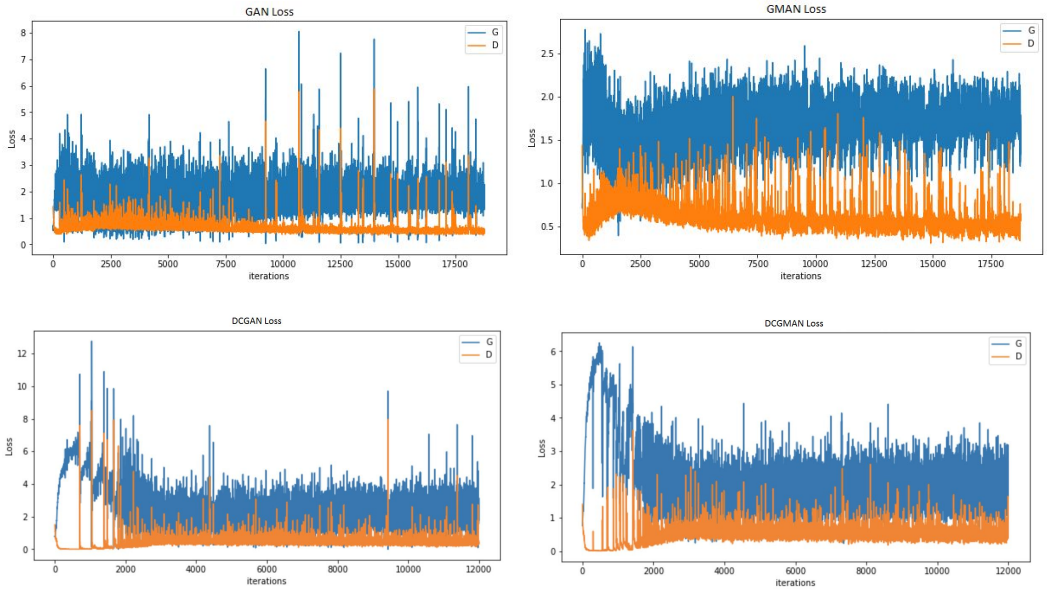


Figure 7: Generator and Discriminator Losses on MNIST data.

GAN **GMAN**

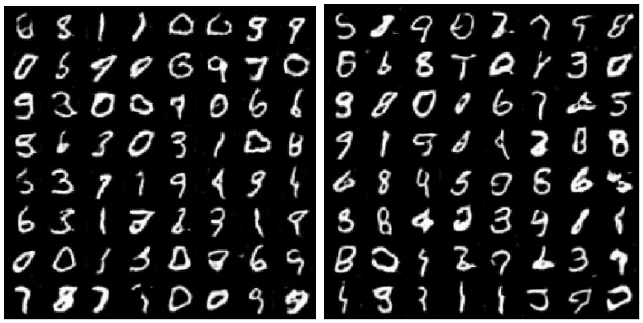


Figure 8: MNIST Outputs of GAN vs GMAN

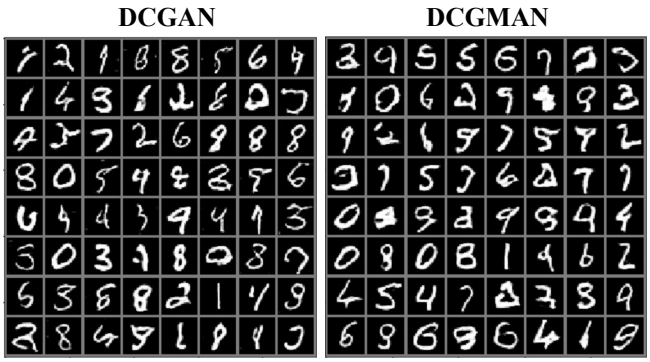
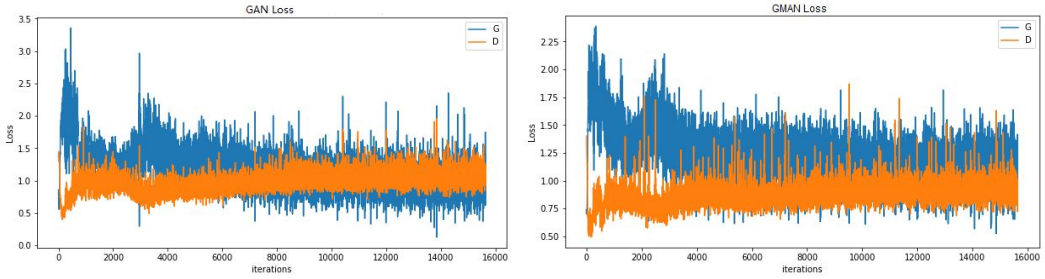


Figure 9: MNIST Outputs of DCGAN vs DCMGAN Models

5.2 CIFAR-10 Outputs

The GAN models are trained for 20 epochs on CIFAR-10 data and the losses of the networks are plotted below in the following figure. Similarly to the MNIST case, the GMAN has more stable training behavior, with fewer spikes in generator and discriminator loss. The outputs of the generators are shown below in the following figures for qualitative comparison only. Again, we use a more objective and quantitative comparison with our generative adversarial metric (GAM).



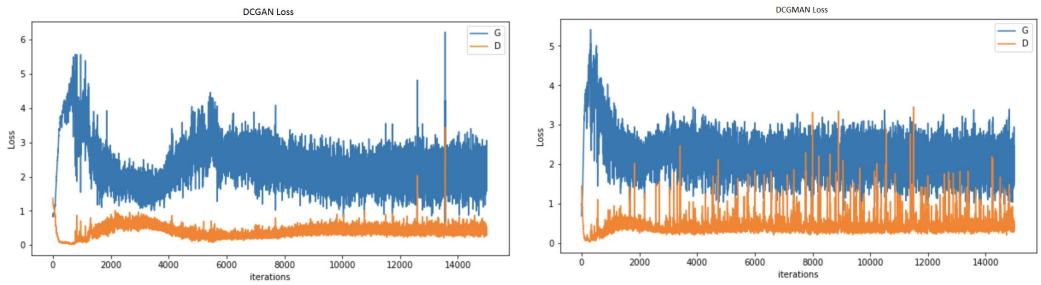


Figure 10: Generator and Discriminator Losses on CIFAR-10 Data. The different color scheme is due to the different development platforms (UCSD Pod Session vs Google CoLab)

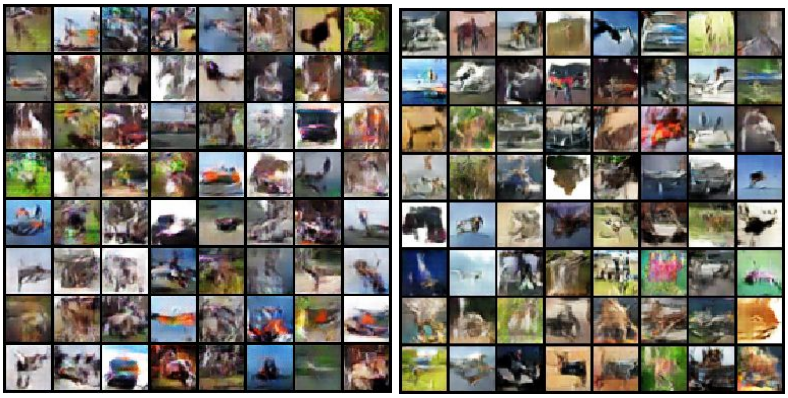


Figure 11: CIFAR-10 Outputs of GAN vs GMAN Models

Figure 12: CIFAR-10 Outputs of DCGAN and DCGMAN Models

5.3 Model Comparisons

Dataset	Model Comparisons	r_{test}	r_{sample}	Winner
MNIST	GAN vs GMAN	1.21	1.02	GMAN
MNIST	DCGAN vs DCGMAN	0.95	1.10	DCGMAN
CIFAR	GAN vs GMAN	0.96	12.7	GMAN
CIFAR	DCGAN vs DCGMAN	1.00	1.37	DCGMAN

Figure 13: Table of GAM results. For both the MNIST and CIFAR Datasets, GMAN and DCGMAN match or exceed single discriminator performance.

6 Discussion

6.1 Inference

This experiment shows that DCGMANs and GMANs are very viable upgrades to the traditional GAN and DCGAN models when generating images. We show a very noticeable trend that the multi-adversarial models consistently outperform (or at match) the original single discriminator models when using these two datasets. The generative adversarial metric validates our qualitative assumptions about their performances from observing their loss and outputs. Adding additional discriminators does indeed enhance overall GAN performance according to the GAM.

6.2 Learning

Doing this project has taught each of us the difficulty in training GANs. However, once we became comfortable experimenting with parameters and using successful models from similar applications, we quickly progressed through the project. We easily surpassed our original proposed goals of comparing GMANs and GANs, using MNIST. To add more depth to our project, and to gain a more meaningful experience from this assignment, we expanded upon our original project proposal and included the CIFAR-10 dataset and DCGANs in our experiments.

6.3 Challenges

A minor challenge is finding a proper fully connected GAN. We initially tried using fully connected networks for the control GANs, but the training process and results were unsatisfactory. Therefore, we added a few convolutional layers to them just to ease the training, but they are still different from the DCGANs described in the paper [5]. The primary challenge in the entire process is finding 5 different discriminator networks for the GMAN. Because feedback from the multi-discriminator network is a mean of all the constituent discriminator networks, using 5 of the same discriminator is too similar to using 1 discriminator. Also, a discriminator that is not paired well with the generator will bring down the performance of the entire network. Although other aggregator functions alleviate this issue better than the mean function can, we still chose to implement it because paper [2] includes many figures generated from the mean function. The authors feature numerical results from using other aggregators, such as weighted mean and softmax, we had difficulty using them meaningfully in our experiments without crucial implementation details (such as how to assign what weights to each network).

6.4 Future Work

Our proposed project idea was to compare GANs and GMANs, and we extended this experiment to also include DCGANs and DCGMANs. Although we also conducted battles between the GANs vs DCGANs and GMANs vs DCGMANs with our existing code infrastructure, they are thematically incompatible with the goals of our experiments. Our goal was to explore the impacts of multiple discriminators only. Therefore, in the near future, we would like to write about the impacts of DCGMANs as an upgrade to GMANs. This would extend our experiments and allow us to explicitly state whether or not deep convolution carries improves GAN performance.

Acknowledgments

Thanks to the ECE285 teaching staff at the University of California San Diego for providing excellent written tutorials and assignments to teach us how to implement neural networks on PyTorch. Thanks to Charles Deledalle, Sneha Gupta, and Shobhit Trehan.

References

[1] Goodfellow, Ian, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. "Generative adversarial nets." In *Advances in neural information processing systems*, pp. 2672-2680. 2014. <GAN>

[2] Durugkar, Ishan, Ian Gemp, and Sridhar Mahadevan. "Generative multi-adversarial networks." arXiv preprint arXiv:1611.01673 (2016). <GMAN>

[3] Im, Daniel Jiwoong, Chris Dongjoo Kim, Hui Jiang, and Roland Memisevic. "Generative Adversarial Metric." arXiv preprint arXiv:1602.05110 (2016). <GAM>

[4] Lecun, Y., Bottou, L., Bengio, Y. & Haffner, P. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86, 2278–2324.

[5] Alex Radford, Luke Metz, Soumith Chintala. “Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks.” arXiv preprint arXiv:1511.06434

[6] Alex Krizhevsky, Vinod Nair & Geoffrey Hinton (). CIFAR-10 (Canadian Institute for Advanced Research)

Appendix:

Github link: https://github.com/houliang428/GAN-for-CIFAR-MNIST-ECE285_Project

The colored and diagrammed documentation and readme are available at the link provided. A text version of readme.md is included below for review:

Generative Multi-Adversarial Neural Networks for Image Generation

Description
This is a final project developed by Jim Jiayi Xu, Zhao Binglin, Xiufeng Zhao, Xiaoyin Yang, Liang Hou for UCSD Fall 2018 ee285 course. Contact us if you have any problem:{jjx002, bzhao, x6zhao, x4yang, l7hou}@eng.ucsd.edu

Prerequisites
We use pytorch and colab/jupyternotebook to train on GPU GTX 1080Ti
Environment:Python3
Below is What you need to install:
```
import os
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import torchvision.datasets as dset
import torchvision.utils as vutils
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
```
Net architectures
For DC1 network, all the layers in the discriminator are deep convolutional layers, and we use transpose convolution for
generator.
For DC2 network, the last convolutional layer is replaced by a fully connected layer.

Training process
Basicly, we use GAN, DCGAN and GMAN as training models and MNIST, CIFAR-10 datasets. (All parameters and architectures are included in our final report, please check [final report](./final_report)

training models and their repository:
For CIFAR-10 dataset:
GAN with GMAN:
[DC1_CIFAR.ipynb](https://github.com/houliang428/ECE285_Project/blob/master/Training/DC1_CIFAR.ipynb)
[DC1_CIFAR.ipynb](https://github.com/houliang428/ECE285_Project/blob/master/Training/DC1_CIFAR_GMAN.ipynb)
DCGAN with GMAN:[DC2_CIFAR.ipynb](https://github.com/houliang428/ECE285_Project/blob/master/Training/DC2_CIFAR.ipynb)

For MNIST dataset:
GAN with GMAN:
[DC1_MNIST.ipynb](https://github.com/houliang428/ECE285_Project/blob/master/Training/DC1_MNIST.ipynb)
[DC1_GMAN_MINST.ipynb](https://github.com/houliang428/ECE285_Project/blob/master/Training/DC1_GMAN_MINST.ipynb)

DCGAN with GMAN:[DC2_MNIST.ipynb](https://github.com/houliang428/ECE285_Project/blob/master/Training/DC2_MNIST.ipynb)

Trained models' parameters
We use torch.save and torch.load to save the already trained models' parameters for furture evaluation.
```
torch.save(netG.state_dict(), 'netX.pt')
netX.load_state_dict(torch.load('netX.pt'))
```
Evaluation method
We use **GAM(Generative adverarial metric)** i.e.battle between GANs to evaluate GAN between their GMAN.
This metric requires two GAN models M1 and M2, each comprised of their constituent G and D networks.

M1={(G1,D1)} and M2={(G2,D2)}

The two stages of using the GAM are the training phase and the testing phase. During the training phase, you train each model by playing the GAN minimax game with its constituent networks. During the testing phase, you compare the performances of each GAN by swapping their D networks. In this testing phase, G1 attempts to trick D2 and G2 attempts to trick D1. (You can see all the equations in our [report](./final_report).)

Demo
1.Download all the files(models,images,training)
2.Upload the files to the jupyter notebook.
3.Run the Demo in notebook with Python3.
(Whole runtime should be less than 30s)

Results