COS 429: Computer Vision

Final Project Report

Generative Adversarial Networks (GAN)

Experimentation & Analysis

Weicong Dong (weicongd@princeton.edu)
Danqi Liao (dl33@princeton.edu)
Xiaorun Wu (xiaorunw@princeton.edu)

December 7, 2020

Contents

1 Introduction & Preambles **3**
- 1.1 Generative Adversarial Network Overview . . . 3
- 1.2 Variations in Generative Adversarial Networks . . . 3
  - 1.2.1 Deep Convolutional GAN . . . 4
  - 1.2.2 Conditional GAN . . . 4
  - 1.2.3 Wasserstein GAN . . . 5
- 1.3 Our Goal . . . 6
- 1.4 Project Outline . . . 6

2 Experiment Set-up **7**
- 2.1 Implementation Details & Process . . . 7
  - 2.1.1 Training GAN is hard . . . 7
  - 2.1.2 Framework & Computing Resources . . . 7
  - 2.1.3 Building and training GANs . . . 7
- 2.2 Generator Architecture . . . 9
- 2.3 Discriminator Architecture . . . 11
- 2.4 Loss Function . . . 13
- 2.5 Data Collection . . . 13
- 2.6 Hyperparameters . . . 13
- 2.7 Evaluation . . . 13

3 Experiment Results **14**
- 3.1 Basic Model-Vanilla GAN . . . 14
  - 3.1.1 MLP architecture on MNIST data. Trained 200 epochs . . . 14
  - 3.1.2 Conv architecture on MNIST data. Trained 150 epochs . . . 16
  - 3.1.3 MLP architecture on CIFAR10 data. Trained 120 epochs . . . 18
  - 3.1.4 Conv architecture on CIFAR10 data. Trained 120 epochs . . . 20
  - 3.1.5 Conv architecture on CelebA data. Trained 80 epochs . . . 22
- 3.2 Deep Convolutional GAN (DCGAN) . . . 24
  - 3.2.1 DCGAN on MNIST. Trained 100 epochs . . . 24
  - 3.2.2 DCGAN on CIFAR10 dataset. Trained 50 epochs . . . 26
  - 3.2.3 DCGAN on CelebA dataset. Trained 50 epochs . . . 27
- 3.3 Conditonal GAN (CGAN) . . . 28
  - 3.3.1 MLP architecture on MNIST data. Trained 150 epochs . . . 28
  - 3.3.2 Conv architecture on CIFAR10 data. Trained 120 epochs . . . 31
- 3.4 Wasserstein GAN (WGAN) . . . 33
  - 3.4.1 MLP architecture on MNIST data. Trained 150 epochs . . . 33
  - 3.4.2 Conv architecture on CIFAR10 data. Trained 140 epochs . . . 35
  - 3.4.3 Conv architecture on CelebA data. Trianed 110 epochs . . . 37
- 3.5 Experiment Conclusion & Remarks . . . 38

4 Theoretical flavor of GAN **39**
- 4.1 Motivations and chapter outline . . . 39
- 4.2 Generating random variables: how the generator in GAN works . . . 40
  - 4.2.1 The inverse transform method . . . 40
  - 4.2.2 Generator Mechanism . . . 40
- 4.3 Loss functions . . . 41
  - 4.3.1 Optimal Value for D . . . 41
  - 4.3.2 Global Optimal and interpretation of loss function . . . 42
  - 4.3.3 Non-Saturating GAN Loss . . . 42
  - 4.3.4 Wasserstein distance . . . 43
  - 4.3.5 Wasserstein distance as GAN loss function . . . 44
- 4.4 Evaluating GANs . . . 44
  - 4.4.1 Inception Score . . . 44
  - 4.4.2 Fréchet Inception Distance (FID) . . . 45

5 Limitations **46**

5.1 Hard to achieve Nash equilibrium 46
5.2 Vanishing Gradient 47
5.3 Mode Collapse 47

6 Concluding Remarks **48**

1 Introduction & Preambles

1.1 Generative Adversarial Network Overview

Generative adversarial networks (GANs)[1] are algorithmic architectures that, in general, use two neural networks, putting one against the other (thus the "adversarial" relationship), to generate new, synthetic instances of data that can pass for real data. They are used widely in image generation, video generation and voice generation.

One neural network, called the generator, generates new data instances, while the other, the discriminator, evaluates them for authenticity; i.e. the discriminator checks whether its input data actually come from the train dataset or are fake samples generated by the generator, whereas the generator tries to fool the discriminator.

For example, suppose one wants to generate hand-written digits similar to images in MNIST dataset. The goal of the discriminator is to distinguish between real MNIST images and fake images generated by the generator. Meanwhile, the goal of the generator is to generate realistic-looking hand-written digits, aiming to fool the discriminator to think of generated images as real samples from the MNIST dataset.

Here are the general overall steps for training GANs:

1. The generator takes in a latent vector and returns an image.
2. This generated images are fed into the discriminator alongside a stream of images taken from the actual, ground-truth dataset.
3. The discriminator takes in both real and fake images and returns some scores for each of them. The scores could be probabilities or value assigned by some value functions and they represent the degree to which the generator think the input images are real images from the train dataset.
4. Compute the losses for both the generator and discriminator, and back-propagate the errors to update the networks' parameters.

A general structure of Generative Adversarial Network might look like:

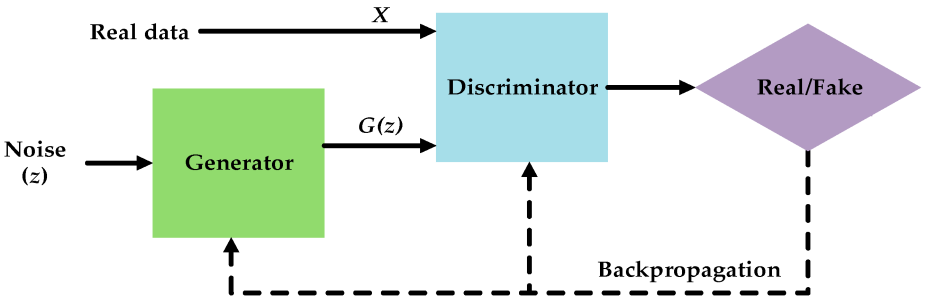


Figure 1: General Structure of GAN

Both nets are trying to optimize a different and opposing objective function, or loss function, in a zero-sum game. As the discriminator changes its behavior, so does the generator, and vice versa. Their losses push against each other.

We will defer the discussion about the training of generative models to chapter 2, and the theoretical analysis of GANs to chapter 4.

1.2 Variations in Generative Adversarial Networks

In the following section, we will give an overview of some of the advancements/developments on the original generative adversarial networks since it first came out in 2014. We will defer the experimentation, discussion and analysis of each model in chapter 2. So in this project, we focus on three variations

of the original GAN (which is also referred to as the Vanilla GAN, where henceforth we will use the term GAN and Vanilla GAN interchanably):

1.2.1 Deep Convolutional GAN

Deep Convolutional GAN (DCGAN)[2] is one of the popular and successful improvement upon original GAN. It mainly composes of convolution layers without max pooling or fully connected layers. It focuses on using multiple convolutional and transposed convolutional layers for the downsampling and the upsampling, and employs techniques such as batch normalization and ReLU in its architecture designs.

The general structural improvement of DCGAN from the original GAN includes the following:

1. Replace all max pooling with convolutional stride
2. Use transposed convolution for upsampling.
3. Eliminate fully connected layers.
4. Use Batch normalization except the output layer for the generator and the input layer of the discriminator.
5. Use ReLU in the generator except for the output which uses tanh.
6. Use LeakyReLU in the discriminator.

A structural demonstration of DCGAN is shown in the image below:

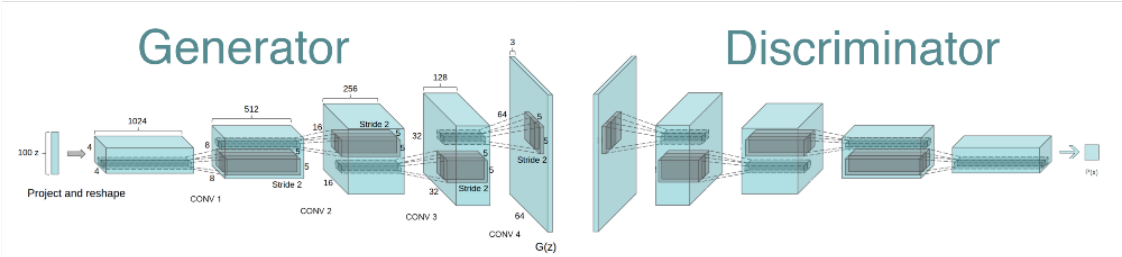


Figure 2: Structural demonstration of Deep Convolutional Generative Adversarial Networks

Again, we will explain the structure, experimental set-up and some of our results of DCGAN in chapter 2.

1.2.2 Conditional GAN

Conditional Generative Adversarial Network (CGAN) is a conditional version of generative adversarial nets. CGAN can be constructed by simply feeding the data on which the input image is conditioned to both the generator and discriminator.

The following image gives a brief outline of the structural difference between Vanilla GAN and CGAN:

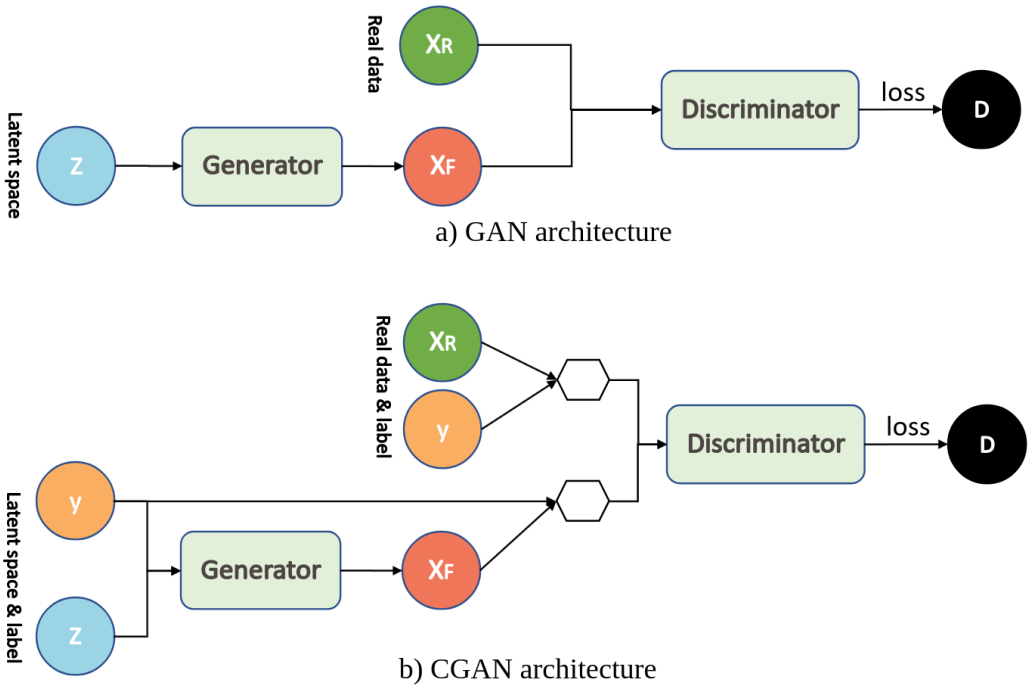


Figure 3: Structural demonstration of Deep Convolutional Generative Adversarial Networks

The following discussion might be slightly technical. We will explain the architectures and implementation details to greater depth in chapter 2

We note that Generative adversarial nets can be extended to a conditional model if both the generator and discriminator are conditioned on some extra information y. Where:

1. y could be any kind of auxiliary information, such as class labels or data from other modalities.
2. The conditioning is performed by feeding y into the both the discriminator and generator as additional input.
3. In the generator the prior input noise $p_z(z)$, and y are combined in joint hidden representation.

So in short, the key difference between Conditonal GAN (CGAN) and vanilla GAN lies within the loss function. The loss function of vanilla GAN is given by:

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

whereas in CGAN, the objective function of a two-player minimax game become:

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

1.2.3 Wasserstein GAN

Wasserstein GAN (WGAN) develops from the GAN. Compared to the vanilla GAN, the WGAN makes the following changes:

1. After every gradient update on the critic function, clamp the weights to a small fixed range, $[-c, c]$.
2. Use a new loss function derived from the Wasserstein distance. The "discriminator" model does not play as a direct critic but a helper for estimating the Wasserstein metric between real and generated data distribution.
3. Empirically the authors recommended RMSProp (which stands for Root Mean Square Propagation) optimizer on the critic, rather than a momentum based optimizer such as Adam which could cause instability in the model training.

The general structure of the algorithm is shown below.

Algorithm 1 WGAN, our proposed algorithm. All experiments in the paper used the default values $\alpha = 0.00005$, $c = 0.01$, $m = 64$, $n_{\text{critic}} = 5$.

Require: : α, the learning rate. c, the clipping parameter. m, the batch size. n_{critic}, the number of iterations of the critic per generator iteration.
Require: : w_0, initial critic parameters. θ_0, initial generator's parameters.
while θ has not converged **do**
  for $t = 0, ..., n_{\text{critic}}$ **do**
    Sample $\{x^{(i)}\}_{i=1}^{m} \sim \mathbb{P}_r$ a batch from the real data.
    Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
    $g_w \leftarrow \nabla_w \left[\frac{1}{m}\sum_{i=1}^{m} f_w(x^{(i)}) - \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))\right]$
    $w \leftarrow w + \alpha \cdot \text{RMSProp}(w, g_w)$
    $w \leftarrow \text{clip}(w, -c, c)$
  end for
  Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
  $g_\theta \leftarrow -\nabla_\theta \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))$
  $\theta \leftarrow \theta - \alpha \cdot \text{RMSProp}(\theta, g_\theta)$
end while

Figure 4: Algorithm of Wasserstein Generative Adversarial Networks

1.3 Our Goal

Our main goal for this final project is three-fold:
Firstly, we aim to understand GAN from an empirical perspective: we aim to code up, build and train GANs by ourselves. By building, training and evaluating GAN and its variants, we aim to understand better about the general experimental set-up, some of the difficulties, challenges in training GAN, and what are some different possible results that we could obtain when training GAN, and what are some difference we could obtain by training different variations of GAN.

Secondly, we aim to study GAN from a more theoretical perspective: in particular, we aim to understand the underlying process of generating random data from a given distribution; also, we aim to better understand how the loss function of generator and the discriminator works, and how does loss functions of different variations of GAN differ; moreover, we will also briefly discuss some metrics to evaluate GAN performance: the inception score, and the Fréchet Inception Distance (FID).

Lastly, we also aim to study some of the potential limitations of GAN: in particular, we will briefly discuss about the issue of mode collapse, and some of the implications of that. To conclude, we will also briefly discuss some of the applications of GAN.

1.4 Project Outline

So our project is outlined as follows:

1. In chapter 2, we outlined our implementation details, experimental setups, including some of the statistics we've used for preliminary testing, as well as some of the architectural details we've used when implemented various models.

2. In chapter 3, we mainly present our training results, as well as some analysis on the obtained result.

3. In chapter 4, we shift our focus to giving a more theoretical overview of GAN. In particular, we will discuss the mechanisms of generator and discriminator and the loss functions.

4. In chapter 5, we will discuss some of the limitations in the current GAN and its variations, and some improvements on this.

2 Experiment Set-up

To gain hands-on experience on training GAN and its variations from scratch and to understand the challenges and limitations of GANs, we have designed four experiments, training vanilla GAN, DCGAN, Conditional GAN and WGAN, and evaluating them on three data sets: MNIST, CIFAR10 and CelebA.

We have trained around 13 models of different GANs in total and ran more than one hundred training scripts to train different GANs and evaluate them on different datasets. In order to better visualize the generator's training progress, we coded up a fixed latent vector and fed the fixed latent vector to the generator during different training epochs.

2.1 Implementation Details & Process

2.1.1 Training GAN is hard

First and foremost, as we were building, experimenting and training GANs, we quickly came to the realization that training GANs is hard and quite finicky as the success of GAN training can be influenced by minor changes in image pre-processing, model architectures and so on, in contrast to training Convolutional Network for image classification where adding and removing layers usually do not change performance drastically.

In addition, the reasons behind why using BatchNorm, LeakyReLU, for example in DCGAN, are more empirically learned instead of theoretically proved, therefore making search for the right generator/discriminator architecture a challenging task.

Adding on top of the finicky nature of GAN training, the lack of clear indication of the training progress is another key reason why GANs are hard to train. In contrast to neural network system used for image classification tasks where loss function directly informs how well the training progresses and how well the trained classifier performs, the loss trajectory of GANs usually does not correlates well with the generated image qualities. For example, during one of our experiments comparing vanilla GAN with MLP architecture and vanilla GAN with Transposed Convolutional/Convolutional architecture, we found that generated CIFAR10 images from MLP architecture has smaller training losses but have actually lower quality than the generated images from Transposed Convolution and Convolution architecture.

Furthermore, training GANs takes much longer iterations and epochs than training neural networks used for image classification tasks. The much longer training time of GANs makes experimenting, iterating, and evaluating much more challenging.

2.1.2 Framework & Computing Resources

We choose PyTorch as our deep learning framework since it builds computational graph dynamically and Google Cloud Compute Instance as our training platform since it usually takes large training epochs to generate decently realistic images using GAN and we need GPUs to make training faster. Being able to train GANs fast is important since GANs takes much longer time to train: more often than not, it takes more than 50 epochs of training iterations for GANs to start generating realistic looking images. We trained and evaluated four GANs on Google Cloud Compute Instance we created with 8 vCPUs, 52 GB memory and one NVIDIA Tesla P100 GPU. It costs around 120 dollars to train and evaluate our four GAN networks on three different datasets.

2.1.3 Building and training GANs

We built each GAN from scratch and tested the models locally on our computer with small epochs to make sure training scripts can be run successfully before we upload them to Google Cloud and train them with much larger epochs.

Taking notes from this DCGAN tutorial in PyTorch[14], we started by building the basic vanilla GAN and training it to generate realistic-looking MNIST images. Since original GAN paper does not specify the architectures used in building the GANs, first we experimented and explored the architectures

of the generator network and discriminator network. The first architectures we tried is a four-layer perception for the generator and LeNet-5 Convolutional Network for the discriminator. We found out that using MLP for generator and Convotional Network for discriminator led to strong discriminator and relatively weak generator within few iterations, and the strong discriminator quickly learns how to differentiate between real images from MNIST and generated images from the generator: the discriminator would assign 1.0 probability to the real images and 0.0 probability to the generated sample after training 1 epoch, in contrast to getting closer to the equilibrium as the original GAN paper described in the ideal situation; on the other hand, because the discriminator becomes too strong very quickly, the generator simply failed to learn how to generate realistic-looking MNIST images.

To prevent the discriminator from getting too strong too quickly, we then experimented with replacing Convolutional Network with four-layer perceptron as the discriminator's architecture. However with four-layer perceptron networks on both the generator and the discriminator, we still failed to successfully train a generator. Inspired by this successful discriminator and generator architectures[13], we then added one layer of LeakyReLU to each fully-connected layer in generator and one layer of LeakyRuLU followed by one layer of dropout to each fully-connected layer in the discriminator. Our intuition about adding extra dropout in the discriminator is that dropout layers make the discriminator harder to become strong quickly, thus gives the generator more time to learn.

Besides the MLP architecture for both the generator and discriminator, we also experimented and explored with Transposed Convolutional Network for the generator and Convolutional Network for the discriminator, since Transposed Convolutional and Convolutional networks are more powerful in capturing the spatial relationships and extracting features of images than linear fully-connected layers. Our experiments empirically show that Transposed Convolutional and Convolutional networks generally generate images with better quality than MLP.

Getting the first vanilla GAN to work on MNIST images took much longer time than we originally thought, as we learnt the hard way that training GAN is difficult. After successfully trained the basic GAN, we moved on to implement and experiment DCGAN. Having the experience of building and training the vanilla GANs with different architectures, We found DCGAN is relatively easier to implement and train since the architectures of generator and discriminator and hyper-parameters are already specified in the paper. We also quickly discovered that DCGAN took much less training epochs for the generator to start generating good quality images.

Besides vanilla GANs and DCGANs, we also implemented and trained Conditional GAN(CGAN). The challenging part of CGAN is to figure out a way to feed both the input class and the latent vector (for generator) or the image (for discriminator) during training. We will discuss the way we merge the conditioned class and input data in the "Generator Architecture" and "Discriminator Architecture" sections.

The last GAN we implemented and explored is Wasserstein GAN(WGAN)[3]. The generator and discriminator architecture of WGAN are similar to those of vanilla GAN. The biggest difference between WGAN and vanilla GAN is the loss function and the training process. We used the Wasserstein Distance loss function and did weight clipping after each update of the discriminator.

By building, training and experimenting different GANs, we discovered three key ingredients on successfully training GANs.

1. Getting the generator and discriminator architecture right. Different train datasets might require different architectures. Minor changes in architectures (such as removing of LeakyReLU layer) can cause big difference in training.

2. The second ingredient, much to our surprise, is getting image pre-processing right. We used torchvision's datasets and transform methods to download and transform input images. To normalize the input image to range [-1,1], we subtracted the mean of the images and divide by the standard deviation, which is easily achieved by `torchvision.transforms.Normalize` method. When we were building our vanilla GAN we accidentally coded up the wrong mean and standard deviation and this pre-processing misalignment caused the whole training process to collapse.

3. Last but not the least, the loss function used in GAN training is also very important. The original GAN paper proved the convergence and conditions of optimality of the minimax loss function, but suggested to use the non-saturating loss function where the generator tries to maximize the log probability of the generated images, assigned by the discriminator. The reasoning is that non-saturating loss function provides sufficient gradient for the generator to learn even when the discriminator assigns 0 probability to the generated samples. For training vanilla GAN, DCGAN and Conditional GAN, we used the non-saturating loss function suggested by the original GAN paper. For training WGAN, we used the Wasserstein Distance as proposed by the WGAN paper.

2.2 Generator Architecture

We have mainly explored four architectures and their variants for the generator. Our latent vector size for the generator is 100.

1. MLP (multi-layer perceptron)

Generator MLP Architecture	
Layer	Activation Size
input	100
100 × 1024 Linear Layer	256
LeakyReLU with 0.2 negative slope	256
256 × 1024 Linear Layer	512
LeakyReLU with 0.2 negative slope	512
512 × 1024 Linear Layer	1024
LeakyReLU with 0.2 negative slope	1024
1024 × flattened image size	flattened image size
Tanh	flattened image size

2. Tranposed Convolutional Network for MNIST, CIFAR10

Generator Transpose Convolutional Architecture for MNIST, CIFAR10 Images	
Layer	Activation Size
input	100×1 × 1
1024×4×4 Transpose Conv with stride 1, pad 0	1024×4×4
BatchNormalization Layer	1024×4×4
LeakyReLU with 0.2 negative slope	1024×4×4
512×4×4 Transpose Conv with stride 2, pad 1	512×8×8
BatchNormalization Layer	512×8×8
LeakyReLU with 0.2 negative slope	512×8×8
256×4×4 Transpose Conv with stride 2, pad 1	256×16×16
BatchNormalization Layer	256×16×16
LeakyReLU with 0.2 negative slope	256×16×16
channels×4×4 Transpose Conv with stride 2, pad 1	channels×32×32
Tanh	channels×32×32

3. Transpose Convolutional Network for CelebA

Generator Transposed Convolutional Architecture for MNIST, CIFAR10 Images	
Layer	Activation Size
input	100×1 × 1
1024×4×4 Transpose Conv with stride 1, pad 0	1024×4×4
BatchNormalization Layer	1024×4×4
LeakyReLU with 0.2 negative slope	1024×4×4
512×4×4 Transpose Conv with stride 2, pad 1	512×8×8
BatchNormalization Layer	512×8×8
LeakyReLU with 0.2 negative slope	512×8×8
256×4×4 Transpose Conv with stride 2, pad 1	256×16×16
BatchNormalization Layer	256×16×16
LeakyReLU with 0.2 negative slope	256×16×16
128×4×4 Transpose Conv with stride 2, pad 1	128×32×32
channels×4×4 Transpose Conv with stride 2, pad 1	channels×64×64
Tanh	channels×64×64

4. Deep Convolutional Network for MNIST, CIFAR10

Deep Convolution Generator Architecture for MNIST, CIFAR10 Images	
Layer	Activation Size
input	100×1 × 1
128×4×4 Transpose Conv with stride 1, pad 0	128×4×4
BatchNormalization Layer	128×4×4
ReLU	128×4×4
64×4×4 Transpose Conv with stride 2, pad 1	64×8×8
BatchNormalization Layer	64×8×8
ReLU	64×8×8
32×4×4 Transpose Conv with stride 2, pad 1	32×16×16
BatchNormalization Layer	32×16×16
ReLU	32×16×16
channels×4×4 Transpose Conv with stride 2, pad 1	channels×32×32
Tanh	channels×32×32

5. Deep Convolutional Network for CelebA

Deep Convolution Generator Architecture for CelebA	
Layer	Activation Size
input	100×1 × 1
512×4×4 Transpose Conv with stride 1, pad 0	512×4×4
ReLU	512×4×4
256×4×4 Transpose Conv with stride 1, pad 0	256×8×8
BatchNormalization Layer	256×8×8
ReLU	256×8×8
128×4×4 Transpose Conv with stride 2, pad 1	128×16×16
BatchNormalization Layer	128×16×16
ReLU	128×16×16
64×4×4 Transpose Conv with stride 2, pad 1	64×32×32
BatchNormalization Layer	64×32×32
ReLU	64×32×32
channels×4×4 Transpose Conv with stride 2, pad 1	channels×64×64
Tanh	channels×64×64

6. MLP for Conditional GAN on MNIST images

   For Conditional GAN on MNIST images, we concatenated the latent vector sampled from Gaussian distribution with the embedding of the input class on which the generation is conditioned and feed the concatenated vector to the MLP network described above. The embedding vector of the input class is obtained from 10×10 trainable embedding lookup table.

7. Transposed Convolutional for Conditional GAN on CIFAR10 images

   For Conditional GAN on CIFAR10 images, we first concatenated the latent vector with the embedding of the input class on which the generation is conditioned and feed the concatenated vector with size 110 to a 110×220 Linear layer. The ouput of the Linear Layer go through a LeakyRuLU layer with 0.2 negative slope and then is fed into the Transpose Convonlutional Network for MNIST, CIFAR10 described above.ed and fed the concatenated vector to the MLP network described above. The embedding vector of the input class is obtained from 10×10 trainable embedding lookup table.

2.3 Discriminator Architecture

We have mainly explored four architectures for the discriminator.

1. MLP (multi-layer perceptron)

Discriminator MLP Architecture	
Layer	Activation Size
flattened image size×1024 Linear Layer	1024
LeakyReLU with 0.2 negative slope	1024
Dropout with 0.3 probability	1024
1024 × 512 Linear Layer	512
LeakyReLU with 0.2 negative slope	512
Dropout with 0.3 probability	512
512 × 256 Linear Layer	256
LeakyReLU with 0.2 negative slope	256
Dropout with 0.3 probability	256
256 × 1 Linear Layer	1
Sigmoid if not WGAN	1

2. Convolutional Network for MNIST, CIFAR10, number of classes is 1 for majority GANs but 10 for CGAN ON MNSIT/CIFAR10 images

Convolution Discriminator Architecture for MNIST/CIFAR10	
Layer	Activation Size
input	3×32×32
256×4×4 Conv with stride 2, pad 1	256×16×16
BatchNormalization Layer	256×16×16
LeakyReLU with 0.2 negative slope	256×16×16
512×4×4 Conv with stride 2, pad 1	512×8×8
BatchNormalization Layer	512×8×8
LeakyReLU with 0.2 negative slope	512×8×8
1024×4×4 Conv with stride 2, pad 1	1024×4×4
BatchNormalization Layer	1024×4×4
LeakyReLU with 0.2 negative slope	1024×4×4
number of classes×4×4 Conv with stride 2, pad 0	number of classes×1×1
Sigmoid if not WGAN	number of classes×1×1

3. Convolutional Network for CelebA

Convolution Discriminator Architecture for CelebA	
Layer	Activation Size
input	3×64×64
128×32×32 Conv with stride 2, pad 1	128×32×32
BatchNormalization Layer	128×32×32
LeakyReLU with 0.2 negative slope	128×32×32
256×4×4 Conv with stride 2, pad 1	256×16×16
BatchNormalization Layer	256×16×16
LeakyReLU with 0.2 negative slope	256×16×16
512×4×4 Conv with stride 2, pad 1	512×8×8
BatchNormalization Layer	512×8×8
LeakyReLU with 0.2 negative slope	512×8×8
1024×4×4 Conv with stride 2, pad 1	1024×4×4
BatchNormalization Layer	1024×4×4
LeakyReLU with 0.2 negative slope	1024×4×4
number of classes×4×4 Conv with stride 2, pad 0	number of classes×1×1
Sigmoid if not WGAN	number of classes×1×1

4. Deep Convolutional Network for MNIST, CIFAR10

Deep Convolution Generator Architecture for MNIST and CIFAR10	
Layer	Activation Size
input	channel×32 × 32
32×4×4 Conv with stride 1, pad 1	32×16×16
LeakyReLU with 0.2 negative slope	32×16×16
64×4×4 Conv with stride 1, pad 1	64×8×8
BatchNormalization Layer	64×8×8
LeakyReLU with 0.2 negative slope	64×8×8
128×4×4 Conv with stride 2, pad 1	128×4×4
BatchNormalization Layer	128×4×4
LeakyReLU with 0.2 negative slope	128×4×4
128×4×4 Conv with stride 1, pad 1	number of classes×1×1

5. Deep Convolutional Network for CelebA

Deep Convolution Generator Architecture for CelebA	
Layer	Activation Size
input	channel×64 × 64
64×4×4 Conv with stride 1, pad 1	64×32×32
LeakyReLU with 0.2 negative slope	64×32×32
128×4×4 Conv with stride 1, pad 1	128×16×16
BatchNormalization Layer	128×16×16
LeakyReLU with 0.2 negative slope	128×16×16
256×4×4 Conv with stride 2, pad 1	256×8×8
BatchNormalization Layer	256×8×8
LeakyReLU with 0.2 negative slope	256×8×8
512×4×4 Conv with stride 2, pad 1	512×4×4
BatchNormalization Layer	512×4×4
LeakyReLU with 0.2 negative slope	512×4×4
number of classes×4×4 Conv with stride 1, pad 1	number of classes×1×1
Sigmoid	number of classes×1×1

6. MLP for Conditional GAN on MNIST images

   For Conditional GAN on MNIST images, we concatenated the flattended image vector with the embedding of the input class on which the generation is conditioned and fed the concatenated vector to the MLP network described above. The embedding vector of the input class is obtained from 10×10 trainable embedding lookup table.

7. Convolutional Network for Conditional GAN on CIFAR10 images

   For Conditional GAN on CIFAR10 images, we first fed the image to the network described below. The output vector of size 1024×1×1 then concatenate with the embedding vector of the input class and the concatenated vector is then fed to a Linear/LeakyReLU/Linear/Sigmoid merge network to ouput the final probability of the input images conditioned on the given input class.The embedding vector of the input class is obtained from 10×10 trainable embedding lookup table.

Convolution Discriminator Architecture for CGAN	
Layer	Activation Size
input	3×32×32
256×4×4 Conv with stride 2, pad 1	256×16×16
BatchNormalization Layer	256×16×16
LeakyReLU with 0.2 negative slope	256×16×16
512×4×4 Conv with stride 2, pad 1	512×8×8
BatchNormalization Layer	512×8×8
LeakyReLU with 0.2 negative slope	512×8×8
1024×4×4 Conv with stride 2, pad 1	1024×4×4
BatchNormalization Layer	1024×4×4
LeakyReLU with 0.2 negative slope	1024×4×4
1024×4×4 Conv with stride 2, pad 0	1024×1×1

2.4 Loss Function

We have used two loss functions for training GANs.

1. Non-saturating loss function, for training vanilla GAN, DCGAN and Conditional GAN.
2. Wasserstein Distance, for training WGAN.

For a detailed, theoretical description of the the non-saturating loss function, as well as the Wasserstein Distance, please refer to section 4.3.3 and 4.3.4.

2.5 Data Collection

We have mainly used three datasets for training and evaluating GANs.

Dataset	Training data size	Object classes
MNIST	60000	10
CIFAR10	50000	10
CelebA	202599	1

2.6 Hyperparameters

Hyper-parameter	Value
learning rate	$2e-4$ for majority, $5e-5$ for WGAN
no. of discriminator update per each update of generator	1 for majority, 5 for WGAN
batch size	100 for majority, 64 for WGAN
optimization method	Adam for majority, RMSProp for WGAN
latent vector size	100
number of epochs	depending on different GANs, but usually around 100

2.7 Evaluation

we coded up a fixed latent vector and fed the fixed latent vector to the generator during different training epochs. Then we can monitor the training progress of the generator and see how it performs

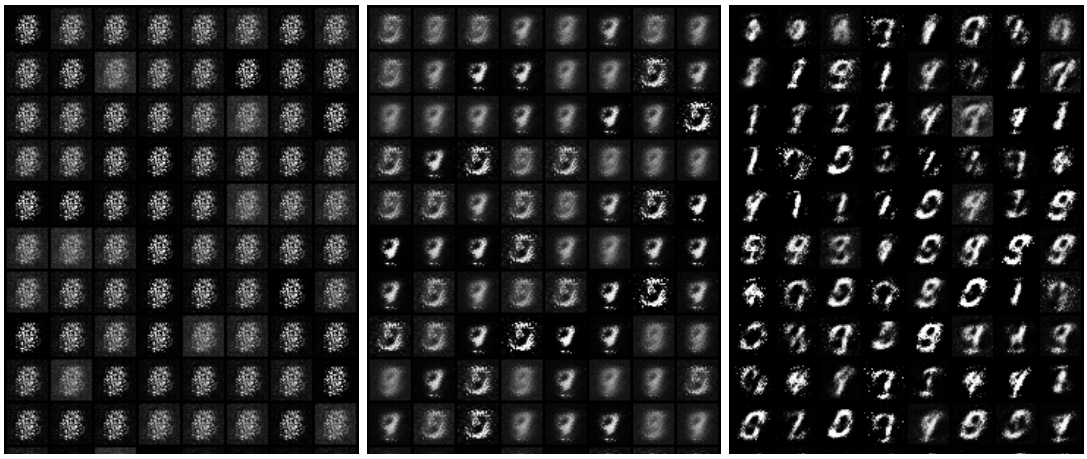
after different epochs by looking at the generated image at different epochs.

It's a well-known challenge to quantitatively evaluate GANs. We considered to use Inception Score[6] to quantitatively analyze the performance of trained generators on various dataset. However, because Inception Score uses Inception v3 model which is trained on 1,000 Object classes of the ILSVRC 2012 dataset[12] and the 1,000 object classes do not contain handwritten digits nor celebrity faces, Inception Score may not be suitable for estimating the generator's performance when the generator is trained to generate MNIST and CelebA images. Only generator trained on CIFAR10 may use Inception Score to estimate its generated image quality and diversity. In addition, all input images to Inception Score method need to be resized to size 299×299, which can be considered as a large scaling for the 32×32 images we trained our GANs to generate. At the end, due to limited time, we decided to gauge the generators' performance by just looking at the generated images by ourselves. Although we did not use it in our experiment, we will discuss the Inception Score from a more theoretical perspective in Chapter 4.

3 Experiment Results

3.1 Basic Model-Vanilla GAN

3.1.1 MLP architecture on MNIST data. Trained 200 epochs



(a) MNIST Epoch 0 (b) MNIST Epoch 5 (c) MNIST Epoch 10

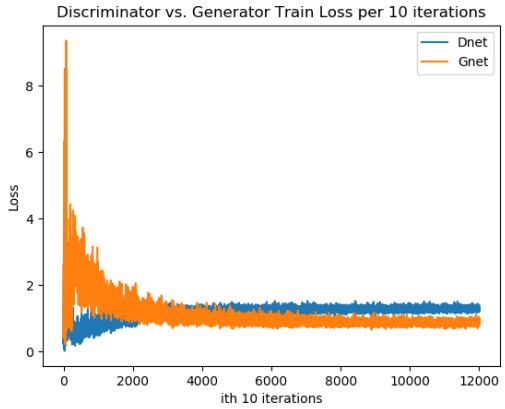
Figure 5: MLP Vanilla GAN on MNIST dataset

(a) MNIST Epoch 30 (b) MNIST Epoch 60 (c) MNIST Epoch 90

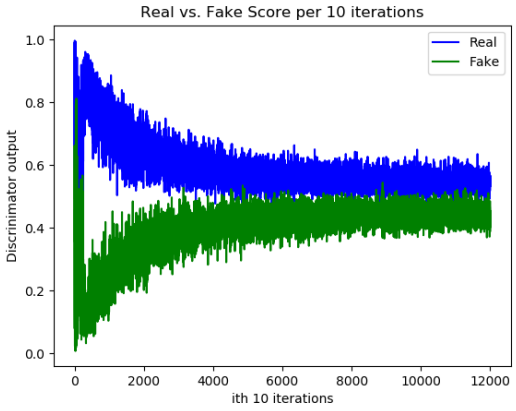
Figure 6: MLP Vanilla GAN on MNIST dataset

(a) MNIST Epoch 120 (b) MNIST Epoch 150 (c) MNIST Epoch 199

Figure 7: MLP Vanilla GAN on MNIST dataset



(a) Train Loss for MLP vanilla GAN on MNIST data



(b) Probability assigned by discriminator

Figure 8: Vanilla GAN training loss and probability assigned by discriminator

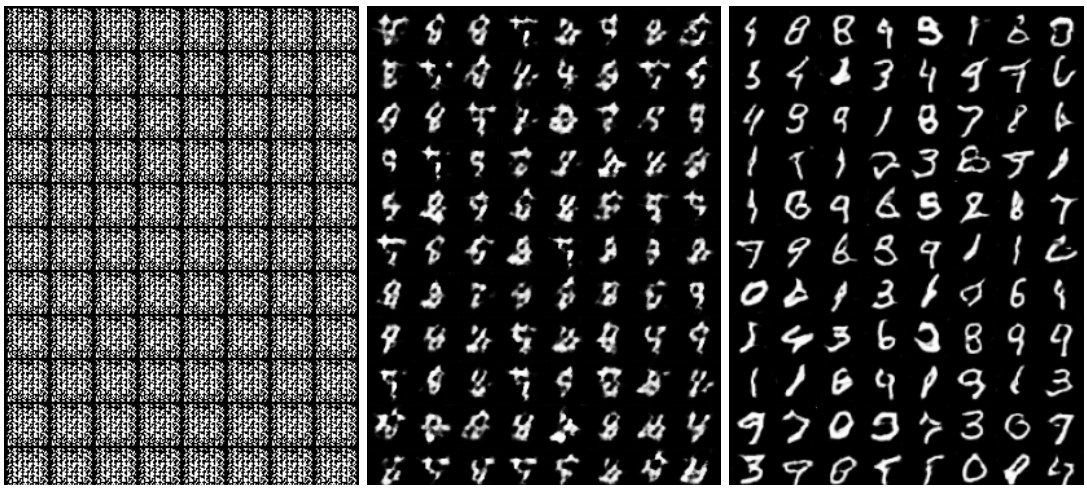
Some remarks about the training:

- Time of Convergence: It takes around 30 epochs for the generated images to start looking like MNIST images, and around 90 epochs for the generated images to look relatively sharp and have less white noise (white dots scattered around the generated images). The generated image quality is still slowly improving after epoch 120, mostly removing the white dots scattered around the

images.

- Generator's train loss starts really high and quickly drops to around 1 2 around 20000 25000 iterations (around 30 epochs). By looking at the width of the plotted line, we can see that the generator loss jitters around more within the initial 30 epochs than after 30 epochs.
- Discriminator's train loss is steadier compared to the generator's. It slowly increases as the train epochs increases and eventually becomes larger than the generator's around 40000 iterations.
- Generator's score, the probability of generated images being real images assigned by the discriminator, starts near 0.0 and converges to 0.4 0.5 after 60000 iterations (around 100 epochs).
- Discriminator's score, the probability for real images being real images assigned by the discriminator, starts near 0.9-1.0 and converges to 0.6-0.5 after 60000 iterations.
- It took us quite a while and couple of failed attempts to get the first basic vanilla GAN to work on MNIST data. We tried with different generator/discriminator architectures. We started with LetNet-5 for the discriminator and MLP for the generator, and also tried MLP with no LeakyRuLU/ReLU nor BatchNormalization layer for both the generator and discriminator. In both cases we failed to train GAN successfully as the discriminator becomes too strong too fast for the generator to have sufficient time to learn.

3.1.2 Conv architecture on MNIST data. Trained 150 epochs

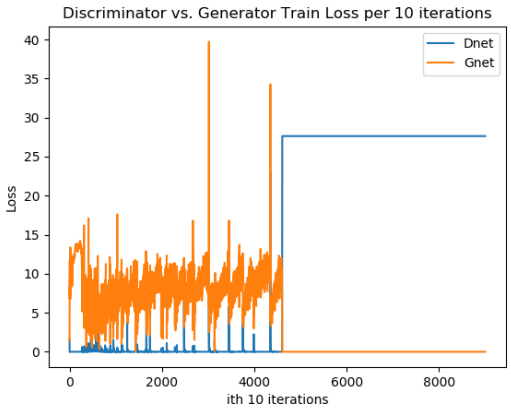


(a) MNIST Epoch 0 (b) MNIST Epoch 5 (c) MNIST Epoch 10

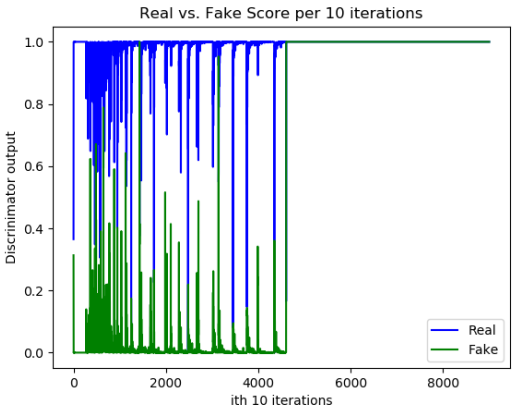
Figure 9: Conv Vanilla GAN on MNIST dataset

(a) MNIST Epoch 30 (b) MNIST Epoch 60 (c) MNIST Epoch 149

Figure 10: Conv Vanilla GAN on MNIST dataset



(a) Train loss for Conv vanilla GAN on MNIST



(b) Probability assigned by discriminator

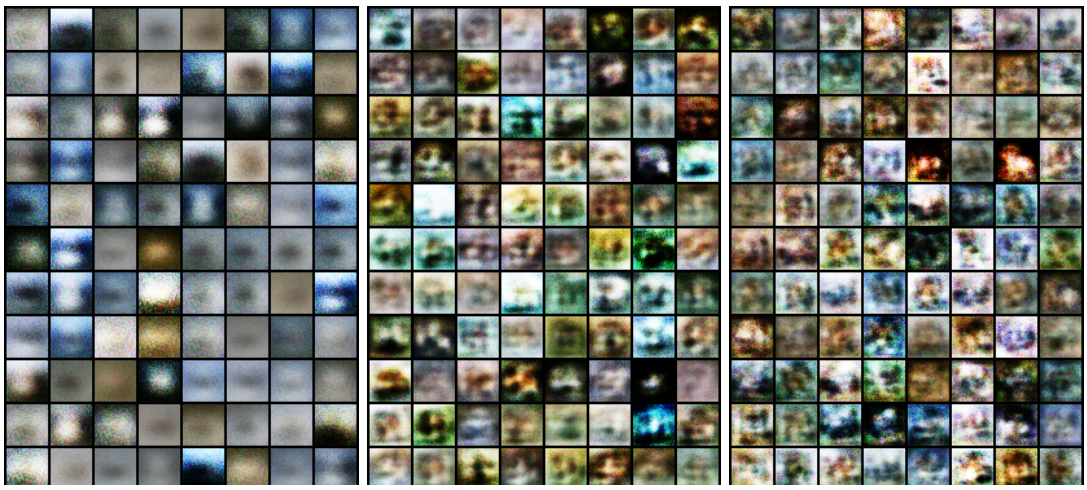
Figure 11: Conv vanilla GAN train loss and probability assigned by discriminator

Some remarks about the training:

- Time of Convergence: The loss and probability oscillate drastically and the generator's loss suddenly drops to zero and its probability shoots up to 1.0 around 45000 iterations (around 75 epochs). The image quality improvement seems trivial after epoch 10.

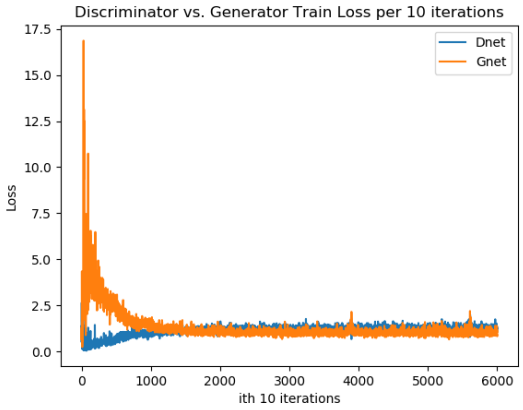
- Conv vanilla GAN converges much faster, in terms of generating realistic looking image, than its counterpart MLP GAN. Conv took around 10 epoch to generate good quality images without white dots whereas MLP GAN took more than 30 epochs and even at 200 epochs still have some white dots scattered in the image.

3.1.3 MLP architecture on CIFAR10 data. Trained 120 epochs

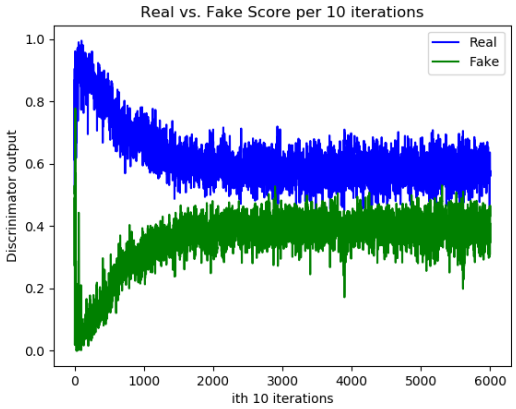


(a) CIFAR10 Epoch 5 (b) CIFAR10 Epoch 50 (c) CIFAR10 Epoch 200

Figure 12: MLP Vanilla GAN on CIFAR10 dataset



(a) Train loss for MLP vanilla GAN on CIFAR10



(b) Probability assigned by discriminator

Figure 13: MLP vanilla GAN train loss and probability assigned by discriminator

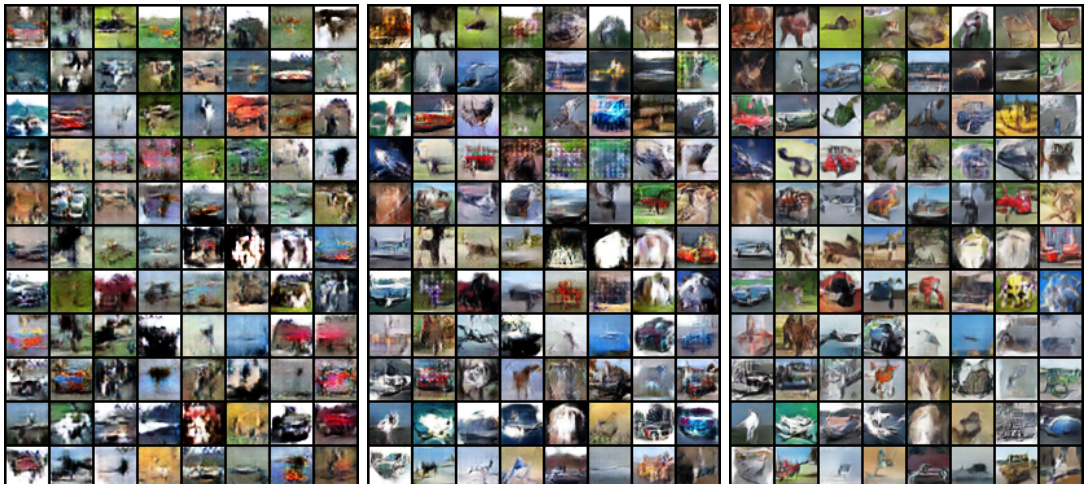
Some remarks on the training:

- Time of Convergence: Generator's train loss decreases and probability increases, and then plateau around 20000 iterations (around 30 epochs). However, the generated images qualities do not improve since they are still blobs of pixels with fuzzy background and shapes even at at epoch 200.

3.1.4 Conv architecture on CIFAR10 data. Trained 120 epochs

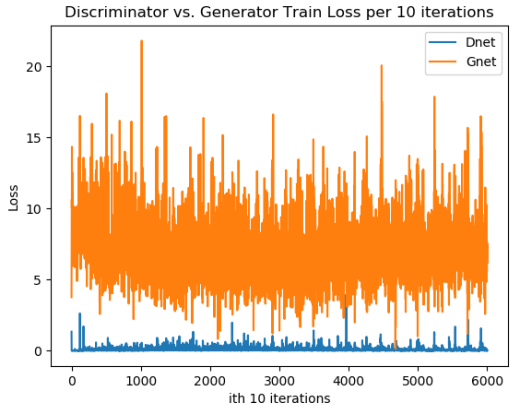
(a) CIFAR Epoch 0 (b) CIFAR Epoch 5 (c) CIFAR Epoch 20

Figure 14: Conv Vanilla GAN on CIFAR dataset

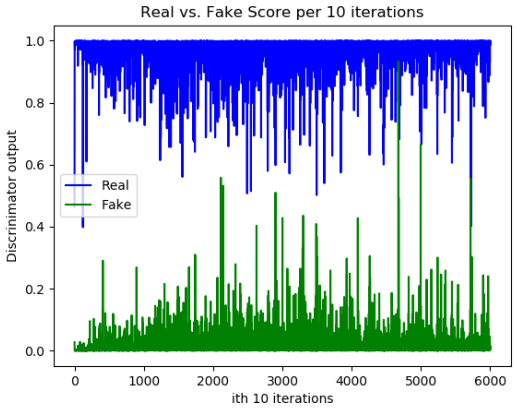


(a) CIFAR Epoch 30 (b) CIFAR Epoch 60 (c) CIFAR Epoch 120

Figure 15: Conv Vanilla GAN on CIFAR dataset



(a) Train loss for Conv vanilla GAN on CIFAR



(b) Probability assigned by discriminator

Figure 16: Conv vanilla GAN train loss and probability assigned by discriminator

Some remarks about the training:

- Time of Convergence: Compared to the MLP vanilla GAN training loss and probability that eventually converge to small range of values, the training loss and probability of Conv vanilla GAN on CIFAR10 oscillates and never converges.
- Despite having oscillating losses, the Conv vanilla GAN is able to generate sharper and better-looking images than the MLP one. There seems to be some red cars, sea animals in generated image of epoch 120. There also seems to be a rooster at the top right corner.

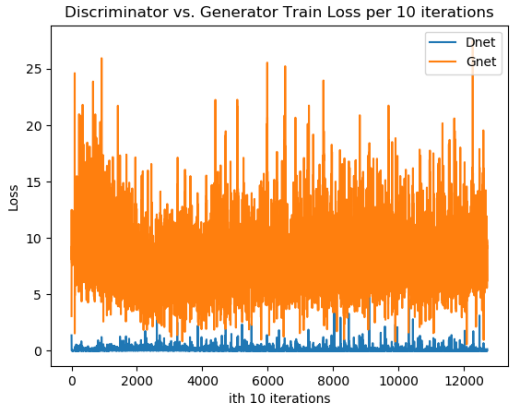
3.1.5 Conv architecture on CelebA data. Trained 80 epochs

(a) CelebA Epoch 1 (b) CelebA Epoch 5 (c) CelebA Epoch 10

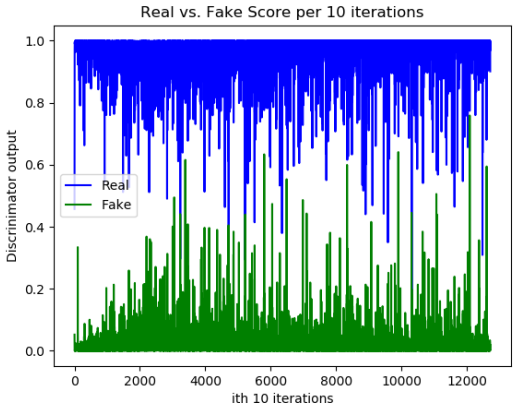
Figure 17: Conv Vanilla GAN on CelebA dataset

(a) CelebA Epoch 30 (b) CelebA Epoch 60 (c) CelebA Epoch 80

Figure 18: Conv Vanilla GAN on CelebA dataset



(a) Train loss for Conv vanilla GAN on CelebA



(b) Probability assigned by discriminator

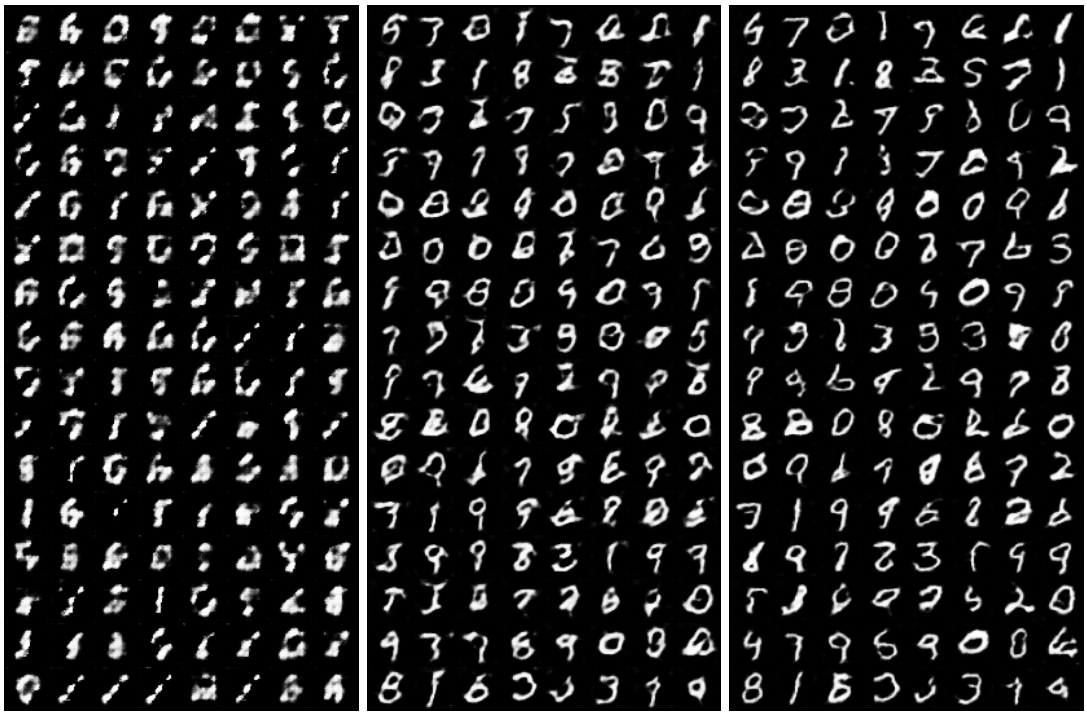
Figure 19: Conv vanilla GAN train loss and probability assigned by discriminator

Some remarks about the training:

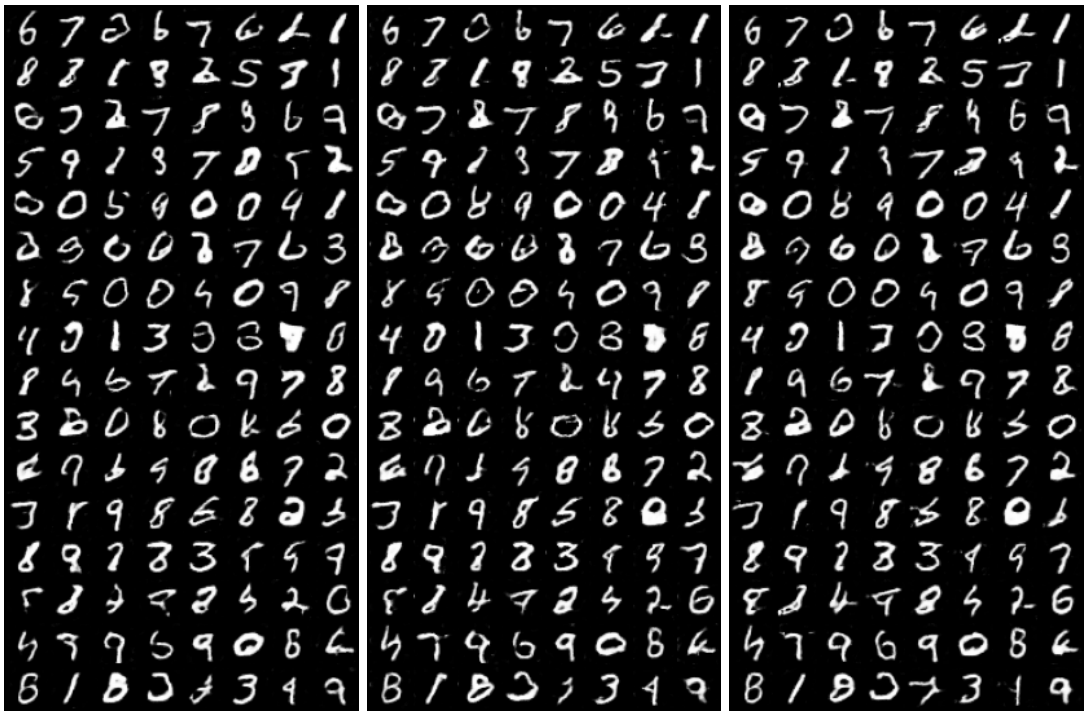
- Time of Convergence: The training loss and probability of Conv vanilla GAN on CelebA oscillates and never converges.
- Despite having oscillating losses, the Conv vanilla GAN is able to generate face like images starting from epoch 10 and the generated face images look sharper as epoch increases. For human eyes, it's till very easily to distinguish the generated faces with real face images since there are blobs and weird deformation around the generated faces.

3.2 Deep Convolutional GAN (DCGAN)

3.2.1 DCGAN on MNIST. Trained 100 epochs

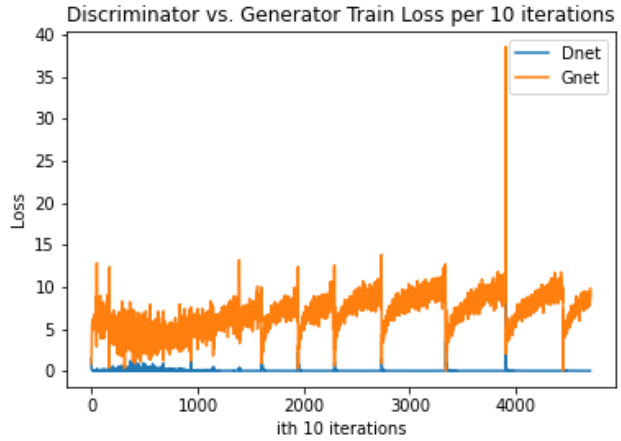


(a) MNIST Epoch 1 (b) MNIST Epoch 5 (c) MNIST Epoch 10

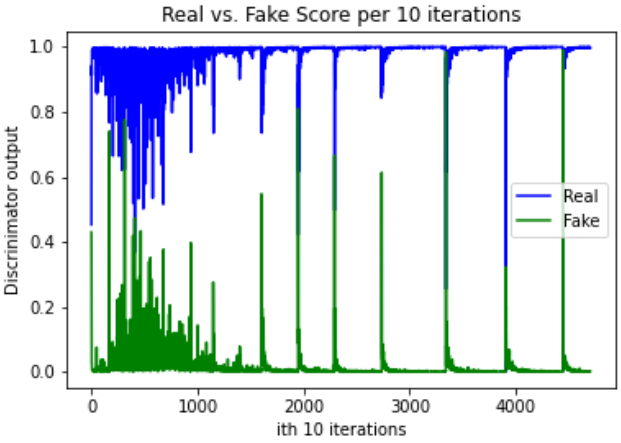


(d) MNIST Epoch 30 (e) MNIST Epoch 60 (f) MNIST Epoch 90

Figure 20: DCGAN on MNIST dataset



(a) Train Loss for DCGAN on MNIST dataset



(b) Probability assigned by discriminator

Figure 21: DCGAN training loss and probability assigned by discriminator

Some remarks about the training:

- Time of Convergence: It takes around 5 epochs for the generator to start generating images similar to MNIST data. After epoch 30, improvement on the quality of generated images is almost trivial.
- Generator's loss oscillates periodically and never converges. Generator's loss increases for a while and then suddenly drops and increase again. When the generator's loss drops, the discriminator's loss has a small spike.
- The probability score of generator changes less drastically before iterations 16000. After iteration 16000, it changes periodically: it shoots up to high value and then immediately drops back near 0.0. It never converges.

3.2.2 DCGAN on CIFAR10 dataset. Trained 50 epochs

(a) Epoch 1 (b) Epoch 3 (c) Epoch 10

(d) Epoch 20 (e) Epoch 30 (f) Epoch 40

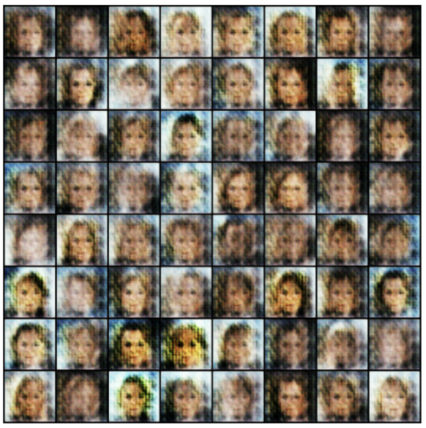
Figure 22: DCGAN on dataset CIFAR10

Figure 23: Train Loss for DCGAN on CIFAR10 dataset

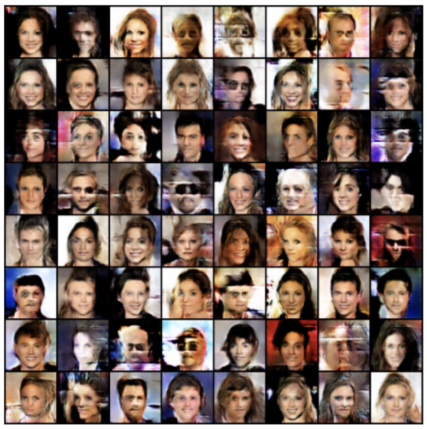
- From epoch 1 to 3, the generated images are pretty blurry.Starting from epoch 10, the generated images begin to remotely look like images from CIFAR10. The generated images become sharper as epoch increases. However even at epoch 40, the generated images look somewhat like CIFAR10 images upon first quick glance but are not realistic objects at all upon closer examination.
- Both of the generator's loss and discriminator's loss oscillate before iteration 26000. Around

iteration 26000 (around epoch 30), the generator's loss shoots up and discriminator's loss becomes 0.

3.2.3 DCGAN on CelebA dataset. Trained 50 epochs



(a) Epoch 1

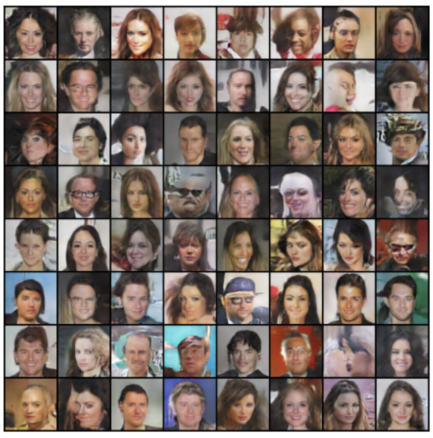


(b) Epoch 5

(c) Epoch 15

(d) Epoch 30

(e) Epoch 40



(f) Epoch 50

Figure 24: DCGAN on dataset CelebA

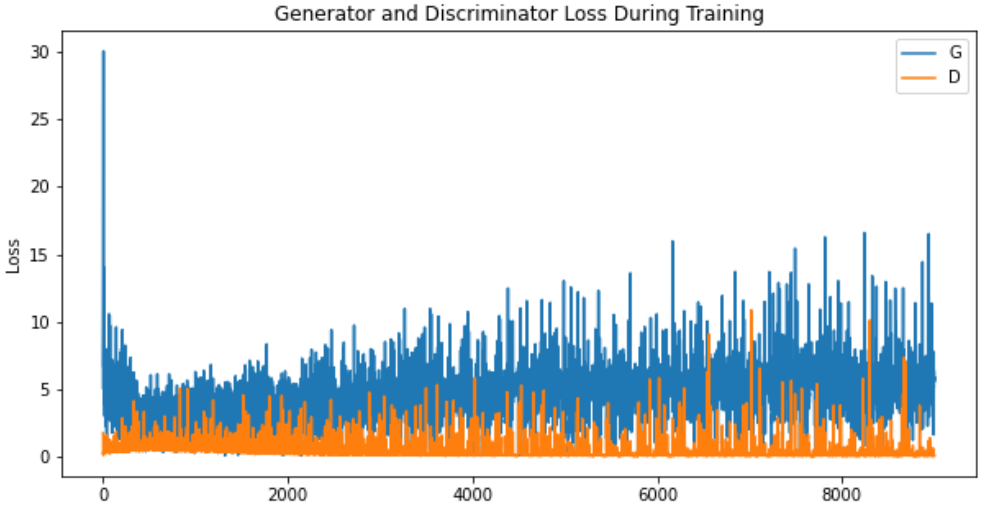
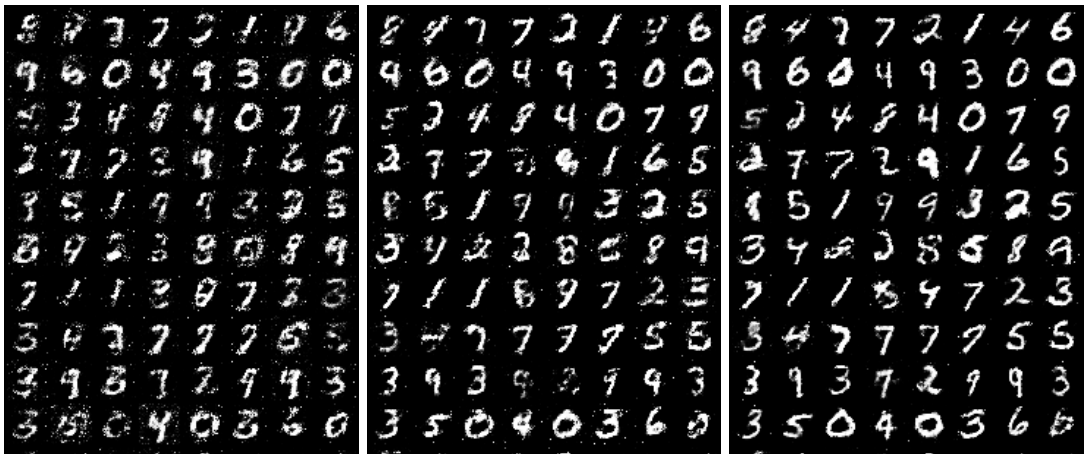


Figure 25: Train Loss for DCGAN on CelebA dataset

- Time of Convergence: The training loss and probability of DCGAN on CelebA oscillates and never converges.
- Despite having oscillating losses, the DCGAN is able to generate face-like images starting around epoch 3-5.The generated face images look sharper as epoch increases. At epoch 50, a small number of the generated faces look fairly realistic.
- It takes much fewer training epochs for DCGAN's generator to generate images remotely have the shape and features of faces (around epoch 3-5) than the vanilla GAN, which takes more than 10 epochs.

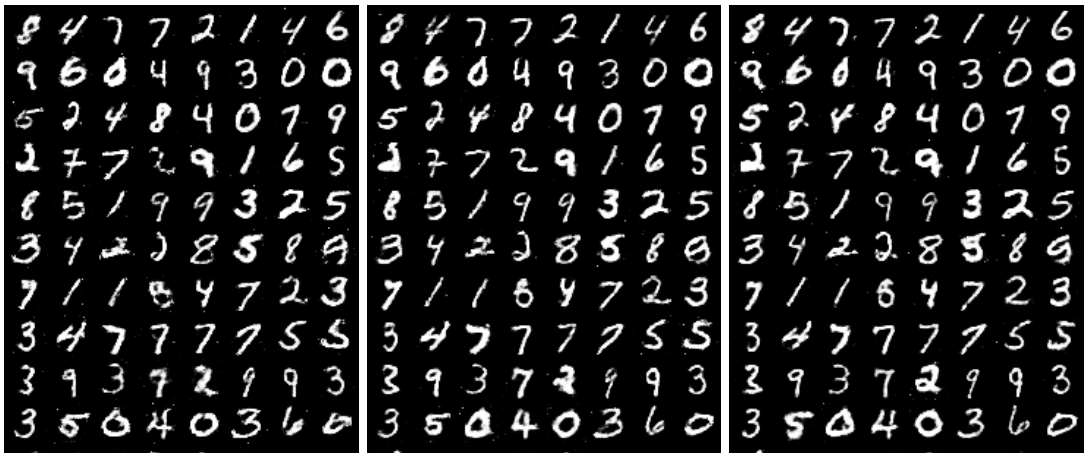
3.3 Conditonal GAN (CGAN)

3.3.1 MLP architecture on MNIST data. Trained 150 epochs



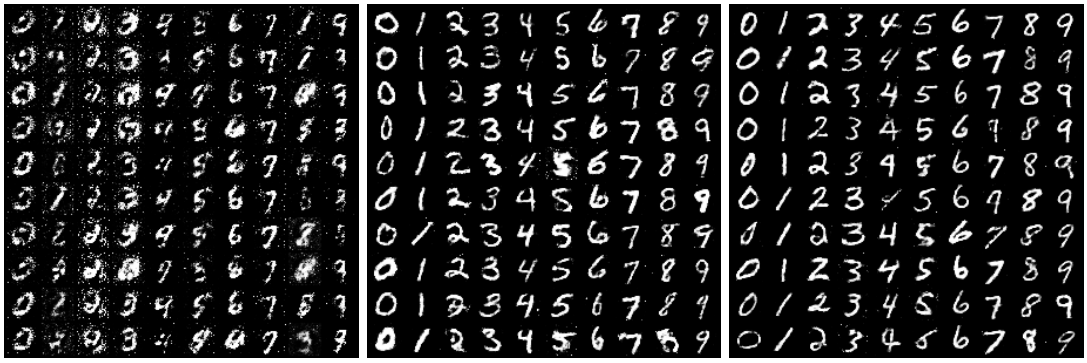
(a) MNIST Epoch 5 (b) MNIST Epoch 10 (c) MNIST Epoch 30

Figure 26: MLP Vanilla CGAN on MNIST dataset



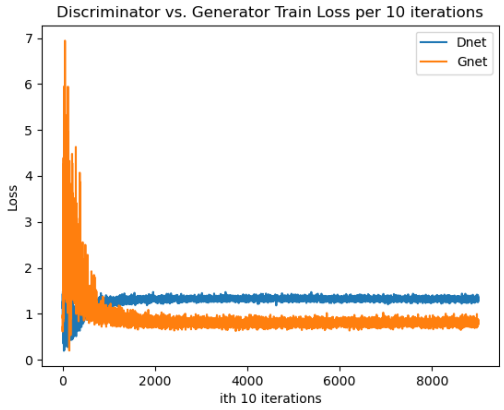
(a) MNIST Epoch 60 (b) MNIST Epoch 100 (c) MNIST Epoch 150

Figure 27: MLP CGAN on MNIST dataset

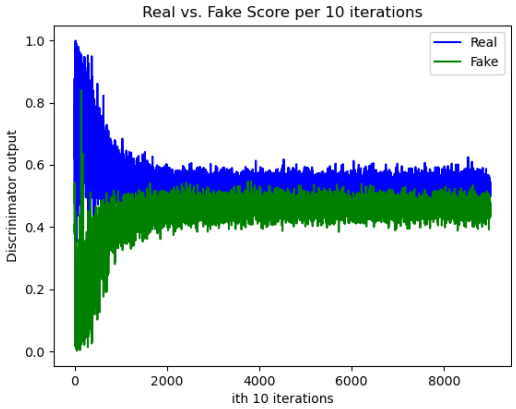


(a) Generated images from class 0 to 9 at epoch 5 (b) Generated images from class 0 to 9 at epoch 60 (c) Generated images from class 0 to 9 at epoch 120

Figure 28: MNIST class 0 to 9 generated by MLP CGAN



(a) Train loss for MLP CGAN on MNIST



(b) Probability assigned by discriminator

Figure 29: MLP CGAN train loss and probability assigned by discriminator

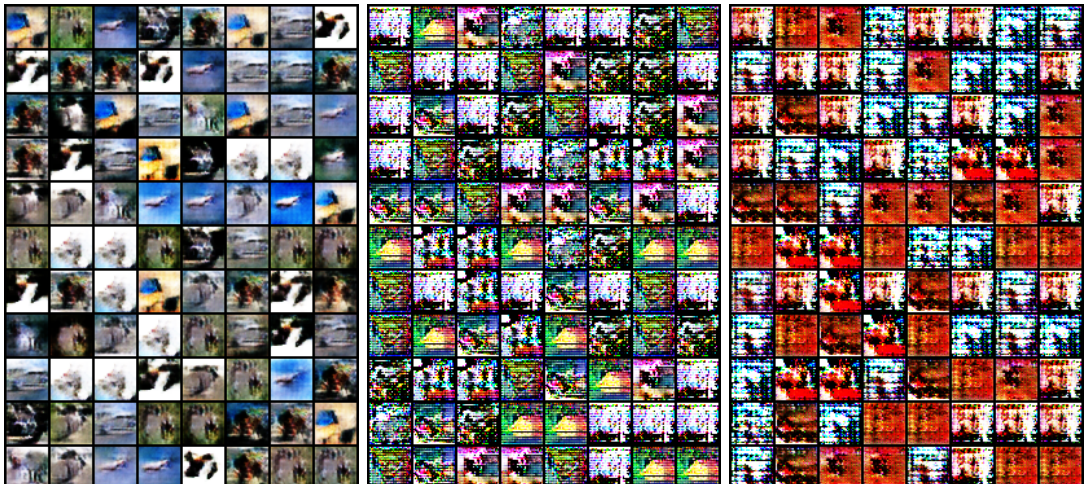
Some remarks about the training:

- Time of Convergence: Generator's loss converges around 16000 iterations (26 epochs), much faster than the vanilla GAN version which takes around 100 epochs.
- Generated images start looking similar to MNIST images starting from epoch 3-5, much faster than the unconditioned vanilla GAN version. Seems like providing more information and have the generator more constrained helps the generator to learn the underlying probability density of the train input data.
- Generator's loss and probability assigned by discriminator still jitters around bit to a lesser extent than the unconditioned vanilla GAN.
- Compared with vanilla GAN at epoch 150, the conditional GAN generates sharper images with less white dots scattered in the image.

3.3.2 Conv architecture on CIFAR10 data. Trained 120 epochs

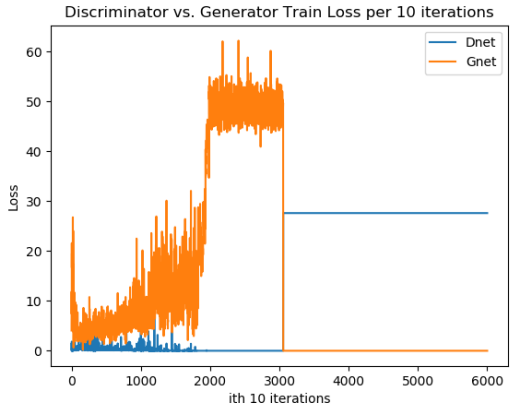
(a) CIFAR Epoch 3 (b) CIFAR Epoch 5 (c) CIFAR Epoch 10

Figure 30: Conv CGAN on CIFAR dataset

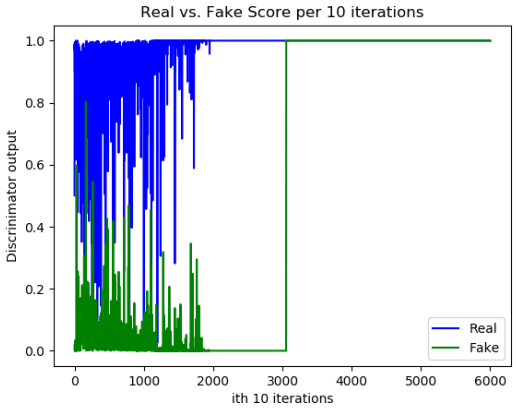


(a) CIFAR Epoch 20 (b) CIFAR Epoch 40 (c) CIFAR Epoch 120

Figure 31: Conv CGAN on CIFAR dataset



(a) Train loss for Conv CGAN on CIFAR



(b) Probability assigned by discriminator

Figure 32: Conv CGAN train loss and probability assigned by discriminator

Some remarks about the training:

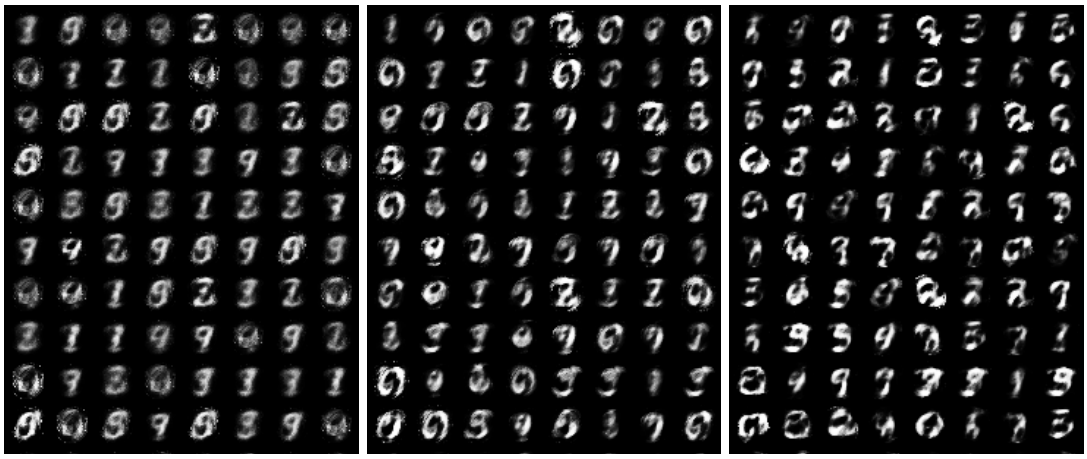
1. Time of Convergence: Before iteration 30000, both generator and discriminator's loss and probability score oscillates widely. Eventually at iteration 30000, the loss and probability score collapsed and generator's loss plummets to 0.0 and probability score shoots up to 1.0, same as the real images' scores.

2. Despite having loss as 0 after 30000 iterations, the generated images start to deteriorate and eventually collapse to noise pattern that are nothing looks like CIFAR10 images from the perspective of humans. The best fake image are around epoch 20-30, where we can vaguely identify some background and figure shapes that are present in CIFAR10 images.

3. Generator's loss decreases while the generated image quality deteriorate. This may suggest that training loss in GAN does not correlate well with the generated image quality.

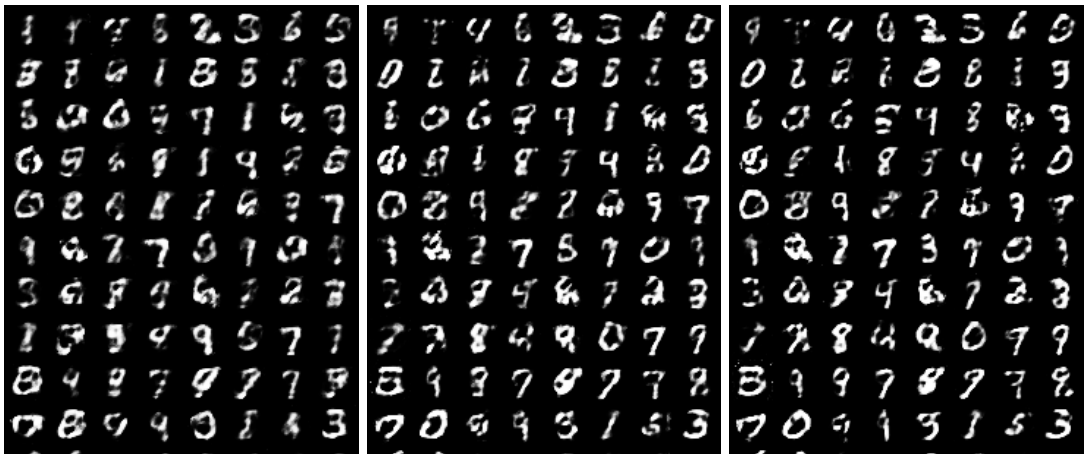
3.4 Wasserstein GAN (WGAN)

3.4.1 MLP architecture on MNIST data. Trained 150 epochs



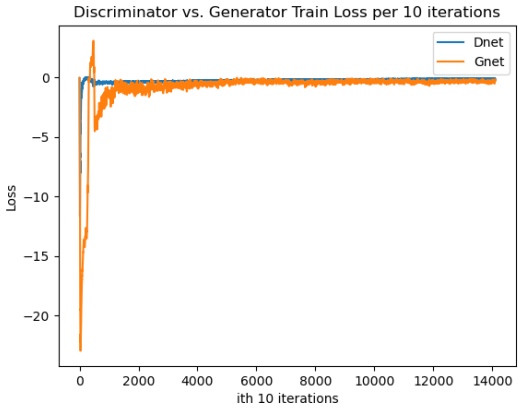
(a) MNIST Epoch 5 (b) MNIST Epoch 10 (c) MNIST Epoch 30

Figure 33: MLP WGAN on MNIST dataset

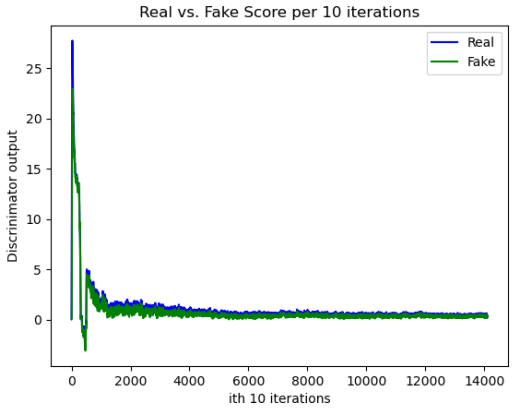


(a) MNIST Epoch 60 (b) MNIST Epoch 100 (c) MNIST Epoch 140

Figure 34: MLP WGAN on MNIST dataset

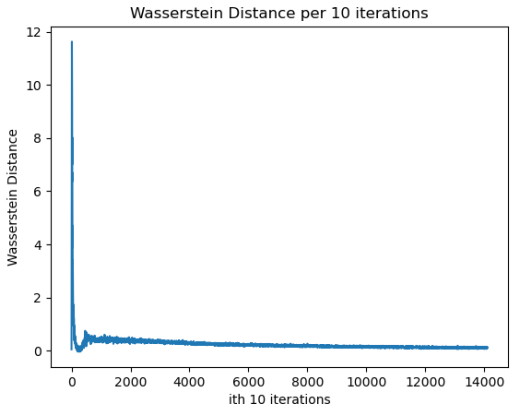


(a) Train loss for MLP WGAN on MNIST



(b) Validity score assigned by discriminator

Figure 35: MLP WGAN train loss and validity score assigned by discriminator



(a) Validity score assigned by discriminator

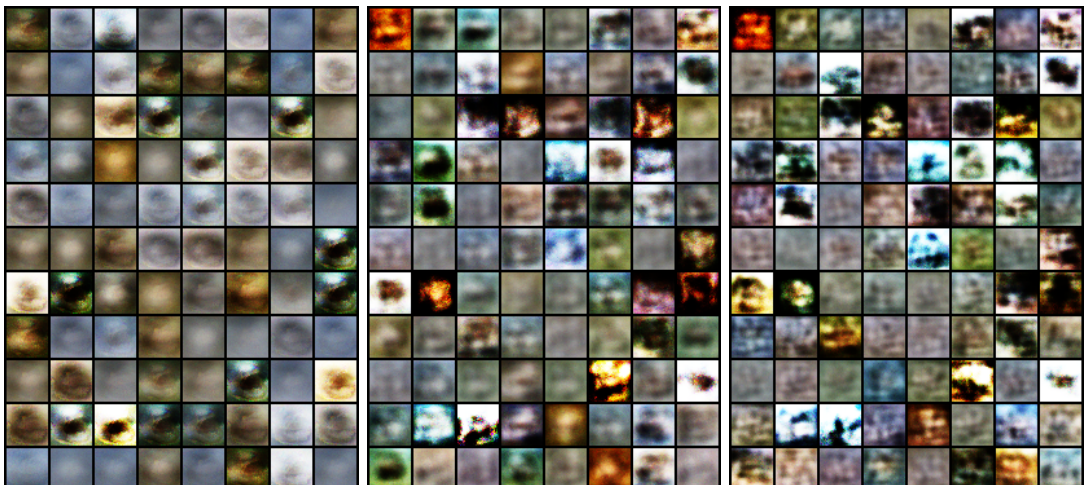
Figure 36: MLP WGAN train loss and validity score assigned by discriminator

Some remarks about the training:

- Time of Convergence: The discriminator's loss quickly jumps to 0 from negative values within 2-3 epochs of training and converges around 0. The generator's loss oscillates a bit within 5 epochs and starts to converge around 0.
- The Wasserstein distance between the real image distribution and generated image distribution starts high and quickly drops near 0.2-0.3 within first 2-3 epochs and then starts converging near 0.

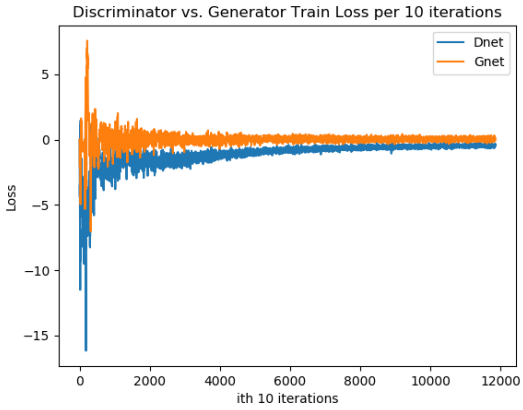
- Despite having generator's loss drops near 0 within 3-5 epochs, the generated images do not quite look like the MNIST images and are very fuzzy. Although the decrease in loss from epoch 5 to epoch 140 is small, we can still see non-trivial improvement in generated image quality.
- Compared with vanilla GAN which takes around 30 epochs to start generating realistic looking MNIST images, WGAN "converges" much slower as it starts generating good-quality images similar to MNIST after epoch 60.

3.4.2 Conv architecture on CIFAR10 data. Trained 140 epochs

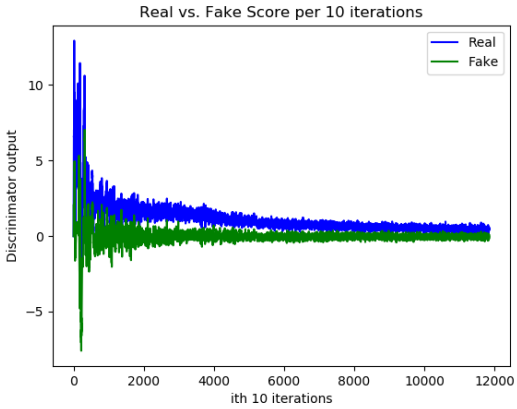


(a) CIFAR Epoch 20 (b) CIFAR Epoch 70 (c) CIFAR Epoch 140

Figure 37: Conv WGAN on CIFAR dataset



(a) Train loss for Conv WGAN on MNIST



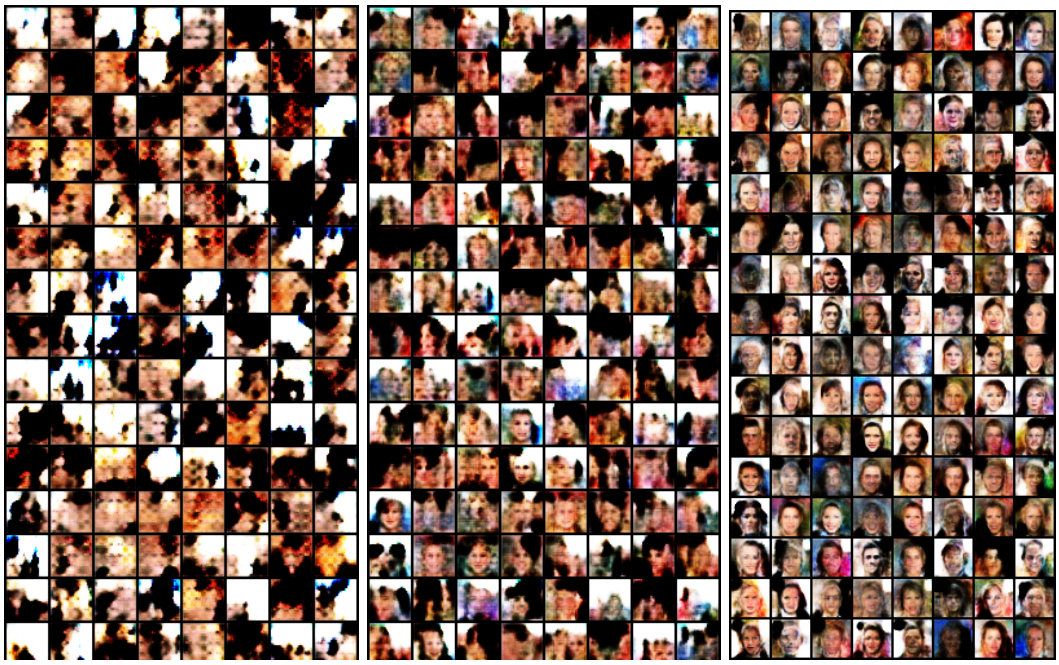
(b) Validity score assigned by discriminator

Figure 38: Conv WGAN train loss and validity score assigned by discriminator

Some remarks on the training:

- Time of Convergence: Discriminator's loss starts in the negative range and quickly increases and converges to 0. Generator's loss starts in the positive range, jitters around drastically during initial epochs and then converges to 0. The validity score assigned by the discriminator to the generated images starts negative and then quickly converges to 0. The validity score of the real images starts high and converges to 0.
- The generator's loss starts to converge around iteration 20000 but the generated images still does not resemble CIFAR10 images.
- Despite having the generator's loss converges to 0, the generated images at the final epoch does not come to be similar to CIFAR10 images.
- WGAN's train loss jitters in a much lesser extent compared to vanilla GANs with the original loss function.

3.4.3 Conv architecture on CelebA data. Trianed 110 epochs

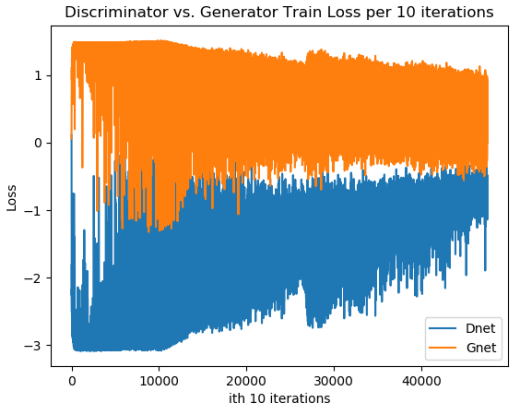


(a) CelebA Epoch 10 (b) CelebA Epoch 20 (c) CelebA Epoch 40

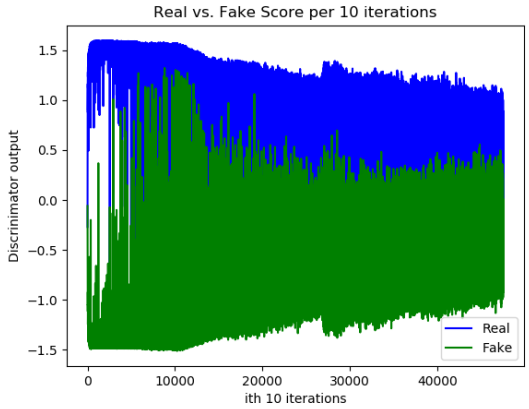
Figure 39: WGAN on CelebA dataset

(a) CelebA Epoch 70 (b) CelebA Epoch 100 (c) CelebA Epoch 140

Figure 40: WGAN on CelebA dataset



(a) Train loss for Conv WGAN on CelebA



(b) Validity score assigned by discriminator

Figure 41: Conv WGAN train loss and validity score assigned by discriminator

Some remarks on the training:

- Time of Convergence: The discriminator's loss, in the general trend, increases as epoch increases. The generator's loss, in general trend, decreases as epochs increases. However, both losses oscillates really drastically between high and low values. The generated images start looking like face images starting around epoch 40 and becomes sharper as epoch increases.
- Compared to Conv GAN with original loss function which starts generating images similar to faces around epoch 10, Conv WGAN "converges" much slower as the generated images start looking like faces around epoch 40.

3.5 Experiment Conclusion & Remarks

1. GANs are hard to train and the success of GANs training is susceptible to minor changes in data pre-processing, generator/discriminator architectures, loss function and so on.
2. GANs usually take a lot of training epochs for the generator to start be able to generate good quality images.
3. DCGAN takes much less epochs for the generator to generate good quality images compared to vanilla GAN, WGAN, CGAN.
4. CGAN takes less epochs for the generator to generate good quality images compared to vanilla GAN, WGAN. GAN is usually faster than WGAN.
5. DCGAN generate the best quality images compared with vanilla GAN, WGAN, CGAN.

6. Generator's training loss does not correlate well with the image qualities generated by the generator, thus making it harder to debug since we couldn't tell if the generator is making progress.

7. Sometimes the generator's loss converges and sometimes it oscillates widely between values. Convergence of the loss does not necessarily guarantee that the generator is able to generate good quality images, and the lack of convergence of loss does not necessarily indicate that the generator is not able to generate good quality images.

8. The training loss of WGAN's generator usually jitters or oscillates in a lesser extent than GANs using the original non-saturating loss function proposed by the original paper.

9. The authors of the WGAN paper suggested that WGAN's loss seems to correlate well with the generated image quality[3]. However in our experiment of WGAN on CIFAR10 data, the train loss decreases but the generated image quality does not strictly increases. In our experiment of WGAN on MNIST data, we see non-trivial improvement in generated image even after the train loss converges.

10. The train losses of GANs trained on CIFAR10 or CelebA are more likely to oscillate than these of GANs trained on MNIST, a much simpler dataset. This suggests that training GANs is heavily dependent on the specific set of images that the generator is trained on.

11. We originally thought it was harder to generate CelebA images than CIFAR10 images. The experiment results showed that generating CIFAR10 is actually much harder. Our intuitive explanation is that CIFAR10 images have more classes and thus the underlying distribution is much more complex than CelebA which are just human faces. quality

4 Theoretical flavor of GAN

4.1 Motivations and chapter outline

With the empirical observations from training the Generative Adversarial Network and its variations in the previous sections, we now shift our focus to discuss some theoretical aspects of it. So why do we pay special attentions to the theoretical aspects of GAN? Our rationale is four-fold:

- First, as we noticed during our training of the various GANs, the behavior and the final output can be fairly unpredictable. As such, a better understanding of the theoretical aspect of it would help us better understand the irregularity of the behavior.

- Second, understanding the theoretical construction would allow us to better fine-tune the parameters.

- Third, it enables us to understand how various models differ from each other from a more systematic way: in particular, we hope that we could quantitatively analyze the difference in various models from the theoretical perspective.

- Last and most importantly, we find it interesting to discover the theoretical aspect behind it, as it linked to many interesting concepts in probability and statistics.

So in particular, we've explored the following 4 aspects of GAN:

1. Random variables generation

2. Loss function across different GAN

3. Evaluating GAN performance: the inception score and the and the Fréchet Inception Distance (FID)

4.2 Generating random variables: how the generator in GAN works

In this section, we discuss the process of generating random variables: we remind some existing methods and more especially the inverse transform method that allows to generate complex random variables from simple uniform random variables, which provides the theoretical foundation for GAN trainning.

In particular, we will be focusing on the inverse transform method, which plays a key role in (vanilla) GAN's generator: since one of the key job of the generator is to start with a random sequence, which we would take a brief diverge to explore how this is achieved.

4.2.1 The inverse transform method

The idea of the inverse transform method is simply to represent our complex random variable as the result of a function applied to a uniform random variable we know how to generate.

Let X be a complex random variable we want to sample from and U be a uniform random variable over $[0, 1]$ we know how to sample from. We also know that the CDF of an random variable is a function from the domain of definition of random variable to the interval $[0, 1]$ and defined, in one dimension, such that

$$\mathrm{CDF}_X)_(x) = \mathbb{P}(X \leq x) \in [0, 1]$$

And for U we know that

$$\mathrm{CDF}_U)_(u) = u, \quad \forall u \in [0, 1].$$

For simplicity, we will suppose here that the function CDF_X is invertible and its inverse is denoted as CDF_X^{-1}.

Then if we define

$$Y = \mathrm{CDF}_X^{-1}(U)$$

then we have

$$\mathrm{CDF}_Y(y) = \mathbb{P}(Y \leq y) = \mathbb{P}(\mathrm{CDF}_X^{-1}(U) \leq y) = \mathrm{CDF}_X(y).$$

As we can see, Y and X have the same CDF and then define the same random variable. So, by defining Y as above (as a function of a uniform random variable) we have managed to define a random variable with the targeted distribution.

Hence inverse transform method allows us to generate a random variable that follows a given distribution by making a uniform random variable goes through a well designed "transform function" (inverse CDF). We also note that this method can be extended to the notion of "transform method" that consists, more generally, in generating random variables as function of some simpler random variables (not necessarily uniform, and the transform function inot necessarily the inverse CDF). Conceptually, the purpose of the "transform function" is to deform/reshape the initial probability distribution: the transform function takes from where the initial distribution is too high compared to the targeted distribution and puts it where it is too low.

4.2.2 Generator Mechanism

So now understanding the method which one may able to generate complex random variables from simple random variables, we would be able to explain who the generator in GAN works. Suppose that we are interested in generating black and white square images of dogs with a size of n by n pixels. We can reshape each data as a $N = n \times n$ dimensional vector. In the same spirit, there exists, over this N dimensional vector space, probability distributions for images of cats, birds and so on.

So using this distributions, the generative network transforms simple random variable into a more complex variable. Then using the targeted distribution as the guideline, it outputs a random variable, which is reshaped to obtain the output image.

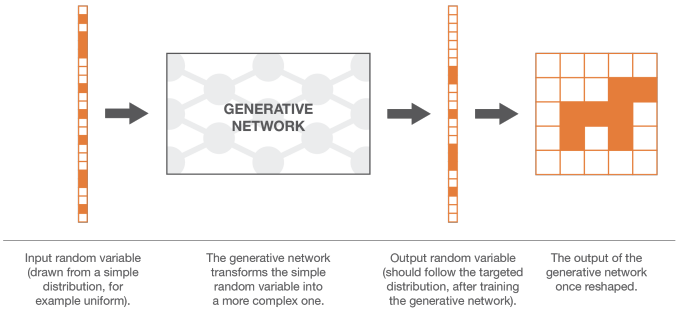


Figure 42: Disribution of p_1 (uniform) and p_2 (Normal)

Now we briefly discuss the loss functions of GAN and WGAN.

4.3 Loss functions

So we know that GAN consists of two models:

- A discriminator D estimates the probability of a given sample coming from the real dataset. It works as a critic and is optimized to tell the fake samples from the real ones.
- A generator G outputs synthetic samples given a noise variable input z (z brings in potential output diversity). It is trained to capture the real data distribution so that its generative samples can be as real as possible, or in other words, can trick the discriminator to offer a high probability.

We also know that D and G compete against each other during the training process: the generator G is trying hard to trick the discriminator, while the critic model D is trying hard not to be cheated. This interesting zero-sum game between two models motivates both to improve their functionalities.

Mathematically, we denote p_z, as the distribution over noise input z. (Usually just uniform); p_g, the generator's distribution over data x, and p_r, the data distribution over the real same x.

On one hand, we want to make sure the discriminator D's decisions over real data are accurate by maximizing $\mathbb{E}_{x\sim p_r(x)}[\log D(x)]$. Meanwhile, given a fake sample $G(z), z \sim p_z(z)$, the discriminator is expected to output a probability, $D(G(z))$, close to zero by maximizing $\mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))]$. On the other hand, the generator is trained to increase the chances of D producing a high probability for a fake example, thus to minimize $\mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))]$.

When combining both aspects together, D and G are playing a minimax game in which we should optimize the following loss function:

$$\min_G \max_D L(D, G) = \mathbb{E}_{x\sim p_r(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))].$$

4.3.1 Optimal Value for D

Now we have a well-defined loss function. Let's first examine what is the best value for D.

$$L(G, D) = \int_x \left(p_r(x) \log D(x) + (p_g(x) \log(1 - D(x))\right),$$

Since we are interested in what is the best value of $D(x)$ to maximize $L(G, D)$, let us label:

$$\tilde{x} = D(x), A = p_r(x), B = p_g(x)$$

And then what is inside the integral (we can safely ignore the integral because x is sampled over all the possible values) is:

$$f(\tilde{x}) = A \log \tilde{x} + B \log(1 - \tilde{x})$$

So that

$$\frac{df(\tilde{x})}{df(\tilde{x})} = \frac{1}{\ln 10} \frac{A - (A + B)\tilde{x}}{x(1 - \tilde{x})}$$

Thus, set $\frac{df(\tilde{x})}{df(\tilde{x})} = 0$, we get the best value of the discriminator:

$$D*(x) = \tilde{x}* = \frac{A}{A+B} = \frac{p_r(x)}{p_r(x)+p_g(x)} \in [0,1].$$

Once the generator is trained to its optimal, p_g gets very close to p_r. When $p_g = p_r$, $D*(x)$ becomes $1/2$.

4.3.2 Global Optimal and interpretation of loss function

When both G and D are at their optimal values, we have $p_g = p_r$ and $D*(x) = 1/2$ and the loss function becomes:

$$L(G, D*) = \int_x (p_r(x)\log D(x) + (p_g(x)\log(1 - D(x))) = \log\frac{1}{2}\int_x p_r(x)dx + \log\frac{1}{2}\int_x p_g(x)dx = -2\log 2.$$

Essentially the loss function of GAN quantifies the similarity between the generative data distribution p_g and the real sample distribution p_r by JS divergence when the discriminator is optimal. The best $G*$ that replicates the real data distribution leads to the minimum $L(G*, D*) = -2\log 2$ which is aligned with equations above.

4.3.3 Non-Saturating GAN Loss

The Non-Saturating GAN Loss is a modification to the generator loss to overcome the saturation problem.It is a subtle change that involves the generator maximizing the log of the discriminator probabilities for generated images instead of minimizing the log of the inverted discriminator probabilities for generated images.

So as we discussed above, the standard GAN loss function, also known as the min-max loss as described in Goodfellow's 2014 paper is

$$\min_G \max_D L(D, G) = \mathbb{E}_{x\sim p_r(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))],$$

The generator tries to minimize this function while the discriminator tries to maximize it. Looking at it as a min-max game, this formulation of the loss seemed effective.

However, in practice it saturates for the generator, meaning that the generator quite frequently stops training if it doesn't catch up with the discriminator.
To address this problem, let's dive deeper in to GAN's loss function. The Standard GAN loss function can further be categorized into two parts: Discriminator loss and Generator loss.

- Discriminator loss
  While the discriminator is trained, it classifies both the real data and the fake data from the generator.

  It penalizes itself for misclassifying a real instance as fake, or a fake instance (created by the generator) as real, by maximizing the below function.

$$\nabla_{\theta_d}\frac{1}{m}\sum_{i=1}^{m}[\log D(\mathbf{x}^{(i)}) + \log(1 - D(G(z^{(i)})))],$$

- Generator loss
  While the generator is trained, it samples random noise and produces an output from that noise. The output then goes through the discriminator and gets classified as either "Real" or "Fake" based on the ability of the discriminator to tell one from the other.

  The generator loss is then calculated from the discriminator's classification – it gets rewarded if it successfully fools the discriminator, and gets penalized otherwise.

  The following equation is minimized to training the generator:

$$\nabla_{\theta_d}\frac{1}{m}\sum_{i=1}^{m}\log(1 - D(G(z^{(i)}))),$$

So to address the problem of saturation, a subtle variation of the standard loss function is used where the generator maximizes the log of the discriminator probabilities $-\log(D(G(z)))$.
This change is inspired by framing the problem from a different perspective, where the generator seeks to maximize the probability of images being real, instead of minimizing the probability of an image being fake. This avoids generator saturation through a more stable weight update mechanism.

In fact, as Goodfellow discussed in NIPS 2016 tutorial, using non-saturated loss is more heuristically motivated: generator can still learn even if discriminator rejects all the sample generated. With this we now turn our attention to Wasserstein distance.

4.3.4 Wasserstein distance

Wasserstein Distance is a measure of the distance between two probability distributions. It is also called Earth Mover's distance, short for EM distance, because informally it can be interpreted as the minimum energy cost of moving and transforming a pile of dirt in the shape of one probability distribution to the shape of the other distribution. The cost is quantified by: the amount of dirt moved x the moving distance.

Formally, suppose we have two distributions, P and Q, each with the probability distribution p_r and p_g respectively, then the earth moving distance is defined as:

$$W(p_r, p_g) = \inf_{\gamma \sim \prod(p_r, p_g)} \mathbb{E}_{(x,y)\sim\gamma}[||x - y||]$$

Where in the formula above $|| \cdot ||$ represents L^2 distance, and $\prod(p_r, p_g)$ is the set of all possible joint probability distributions between pr and pg. One joint distribution $\gamma \in \prod(p_r, p_g)$ describes one dirt transport plan, same as the discrete example above, but in the continuous probability space. Precisely $\gamma(x, y)$ states the percentage of dirt should be transported from point x to y so as to make x follows the same probability distribution of y. That's why the marginal distribution over x adds up to p_g, $\Sigma_x\gamma(x, y) = p_g(y)$ (Once we finish moving the planned amount of dirt from every possible x to the target y, we end up with exactly what y has according to p_g.) and vice versa $\Sigma_y\gamma(x, y) = p_r(x)$.

When treating x as the starting point and y as the destination, the total amount of dirt moved is $\gamma(x, y)$ and the travelling distance is $||x - ||$ and thus the cost is $\gamma(x, y) \cdot ||x - y||$. The expected cost averaged across all the (x, y) pairs can be easily computed as:

$$\sum_{x,y} \gamma(x, y)||x - y|| = \mathbb{E}_{x,y\sim\gamma}||x - y||,$$

Finally, we take the minimum one among the costs of all dirt moving solutions as the EM distance. In the definition of Wasserstein distance, the inf (infimum, also known as greatest lower bound) indicates that we are only interested in the smallest cost.

We note that Wasserstein divergence is a better alternative to the KL divergence as used in the original GAN, since even when two distributions are located in lower dimensional manifolds without overlaps, Wasserstein distance can still provide a meaningful and smooth representation of the distance in-between. This is because Wasserstein metric provides a smooth measure, which is super helpful for a stable learning process using gradient descents. The specific mathematical proof for this is slightly too technical, which we decide to skip here in the constraint of time.

Suppose this function f comes from a family of K-Lipschitz continuous functions, $\{f_w\}w \in W$, parameterized by w. In the modified Wasserstein-GAN, the "discriminator" model is used to learn w to find a good f_w and the loss function is configured as measuring the Wasserstein distance between p_r and p_g.

$$L(p_r, p_g) = W(p_r, p_g) = \max_{w\in W} \mathbb{E}_{x\sim p_r}[f_w(x)] - \mathbb{E}_{z\sim p_r(z)}[f_w(g_\theta(z))],$$

Thus the "discriminator" is not a direct critic of telling the fake samples apart from the real ones anymore. Instead, it is trained to learn a K-Lipschitz continuous function to help compute Wasserstein distance. As the loss function decreases in the training, the Wasserstein distance gets smaller and the generator model's output grows closer to the real data distribution.

4.3.5 Wasserstein distance as GAN loss function

It is intractable to exhaust all the possible joint distributions in $\prod(p_r, p_g)$ to compute $\inf_{\gamma \sim \prod(p_r, p_g)}$. Thus the authors proposed a smart transformation of the formula based on the Kantorovich-Rubinstein duality to:

$$W(p_r, p_g) = \frac{1}{K} \sup_{||f||_L \leq K} \mathbb{E}_{x \sim p_r}[f(x)] - \mathbb{E}_{x \sim p_g}[f(x)],$$

where sup (supremum) is the opposite of inf (infimum); we want to measure the least upper bound or, in even simpler words, the maximum value.

Now define K-Lipschitz continuous as follows: A real-valued function $f : \mathbb{R} \to \mathbb{R}$ is called K-Lipschitz continuous if there exists a real constant $K \geq 0$ such that, for all $x_1, x_2 \in \mathbb{R}$,

$$|f(x_1) - f(x_2)| \leq K|x_1 - x_2|,$$

where K is known as a Lipschitz constant for function $f(\cdot)$.

So the function f in the new form of Wasserstein metric is demanded to satisfy $||f||_L \leq K$, meaning it should be K-Lipschitz continuous.

There are some intersting materials about the transformation behind the Wasserstein distance formula, but due to the technicality, we would omit it here. The details may be checked here: `https://arxiv.org/pdf/1701.07875.pdf`

4.4 Evaluating GANs

In GANs, the objective function for the generator and the discriminator usually measures how well they are doing relative to the opponent. For example, we measure how well the generator is successful in "fooling" the discriminator. It is not a good metric in measuring the image quality or its diversity.

So how exactly do we measure the performance of a particular GAN? Empirically, we've already seen that some GANs "converges" and obtains better result faster than other GANs: for example, the DCGAN seems to generate better images after around 15 epochs, whereas the vanilla GAN takes roughly 50 epochs to generate images of the same quality. Therefore as part of the process for us to understand better about GAN series, we look into the Inception Score and Fréchet Inception Distance on how to compare results from different GAN models.

4.4.1 Inception Score

Inception Score (Henceforth IS) uses two criteria in measuring the performance of GAN:

- The quality of the generated images,
- Their diversity.

The inception score is calculated by first using a pre-trained Inception v3 model to predict the class probabilities for each generated image.

These are conditional probabilities, e.g. class label conditional on the generated image. Images that are classified strongly as one class over all other classes indicate a high quality. As such, the conditional probability of all generated images in the collection should have a low entropy.

So what exactly is entropy? Entropy can be viewed as randomness. If the value of a random variable x is highly predictable, it has low entropy. On the contrary, if it is highly unpredictable, the entropy is high. For example, in the figure below, we have two probability distributions $p(x)$. p_2 has a higher entropy than p_1 because p_2 has a more uniform distribution and therefore, less predictable about what x is.

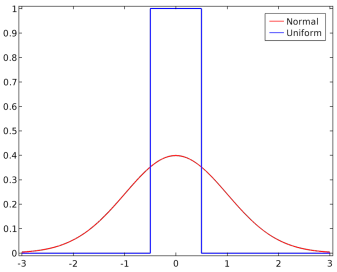


Figure 43: Disribution of p_1 (uniform) and p_2 (Normal)

So mathematically, entropy is calculated as the negative sum of each observed probability multiplied by the log of the probability. The intuition here is that large probabilities have less information than small probabilities. Therefore if we denote the probability of class i as p_i, then the entropy is given by:

$$\text{entropy} = \sum_i (p_i \cdot \log(p_i))$$

To capture our interest in a variety of images, we use the marginal probability. This is the probability distribution of all generated images. We, therefore, would prefer the integral of the marginal probability distribution to have a high entropy. Equivalently, we want the conditional probability $\mathbb{P}(y|x)$ to be highly predictable (low entropy). i.e. given an image, we should know the object type easily. So we use an Inception network to classify the generated images and predict $\mathbb{P}(y|x)$ — where y is the label and x is the generated data. This reflects the quality of the images. Here we note that $\mathbb{P}(y)$ is the marginal probability computed as:

$$\int_z p(y|x = G(z))dx$$

If the generated images are diverse, the data distribution for y should be uniform (high entropy).

Next, we need to measure the diversity of images. These elements are combined by calculating the Kullback-Leibler divergence, or KL divergence (relative entropy), between the conditional and marginal probability distributions.

Calculating the divergence between two distributions is written using the "||" operator, therefore we can say we are interested in the KL divergence between C for conditional and M for marginal distributions or:

$$D_{KL}(C||M) = \int_x p(x) \log \frac{p(x)}{q(x)} dx$$

Specifically, we are interested in the average of the KL divergence for all generated images. Thus putting everything together, we compute their KL-divergence and use the equation below to compute IS as:

$$IS(G) = \exp\left(\mathbb{E}_{\mathbf{x}\sim p_a} D_{KL}(p(y|\mathbf{x})||p(y))\right),$$

While IS provides a good metric to measure both the diversity and the the quality of generated images, it has some limitations as well: One shortcoming for IS is that it can misrepresent the performance if it only generates one image per class. $p(y)$ will still be uniform even though the diversity is low. One alternative approach in evaluating GAN is the Fréchet Inception Distance (FID).

4.4.2 Fréchet Inception Distance (FID)

The FID is supposed to improve on the IS by actually comparing the statistics of generated samples to real samples, instead of evaluating generated samples in a vacuum.

In FID, we use the Inception network to extract features from an intermediate layer. Then we model the data distribution for these features using a multivariate Gaussian distribution with mean μ and covariance Σ. The FID between the real images x and generated images g is computed as:

$$\text{FID} = ||\mu_r - \mu_g||^2 + Tr\left(\Sigma_r + \Sigma_g - 2(\Sigma_r\Sigma_g)^{1/2}\right),$$

where $||\cdot||$ denote the L^2 norm (Euclidean distance), and Tr denote the trace of the resulting matrix, and that $X_r \sim \mathcal{N}(\mu_r, \sigma_r)$ and $X_g \sim \mathcal{N}(\mu_g, \sigma_g)$ are the 2048-dimensional activations of the Inception-v3 pool3 layer for real and generated samples respectively. Similar to IS, Lower FID is better, corresponding to more similar real and generated samples as measured by the distance between their activation distributions.

One advantage of FID is that FID is more robust to noise than IS. If the model only generates one image per class, the distance will be high (hence the IS would likely to be inaccurate). So FID is a better measurement for image diversity. FID has some rather high bias but low variance. By computing the FID between a training dataset and a testing dataset, we should expect the FID to be zero since both are real images. However, running the test with different batches of training sample shows none zero FID.

As we've seen in the above discussion the inception score and the FID provides us with a powerful method to do the evaluate about the performance of GANs. As we mentioned in the previous section, we did not get a chance to try out on these methods to evaluate our model, due to the specificity of IS and FID: they would require the inception v3 model, this means that it is most suitable for 1,000 object types used in dataset such as the ILSVRC 2012, dataset, and might not be suitable in our case.

We now turn our attention to some of the potential drawbacks of the GAN models, such as mode collapse.

5 Limitations

Although GAN has shown great success in the realistic image generation, the training is not easy; The process is known to be slow and unstable. In this section, using the paper by Arjovsky and Bottou, 2017 as our main reference, we will explore some main drawbacks of GAN.

5.1 Hard to achieve Nash equilibrium

Salimans et al. (2016) discussed the problem with GAN's gradient-descent-based training procedure. Two models are trained simultaneously to find a Nash equilibrium to a two-player non-cooperative game. However, each model updates its cost independently with no respect to another player in the game. Updating the gradient of both models concurrently cannot guarantee a convergence.

Let's check out a simple example to better understand why it is difficult to find a Nash equilibrium in an non-cooperative game. Suppose one player takes control of x to minimize $f_1(x) = xy$, while at the same time the other player constantly updates y to minimize $f_2(y) = -xy$.
Because $\frac{\partial f_1}{\partial x} = y$ and $\frac{\partial f_2}{\partial y} = -x$, we update x with $x - \eta \cdot y$ and y with $y + \eta \cdot x$ simultaneously in one iteration, where η is the learning rate. Once x and y have different signs, every following gradient update causes huge oscillation and the instability gets worse in time, as shown in the figure below:

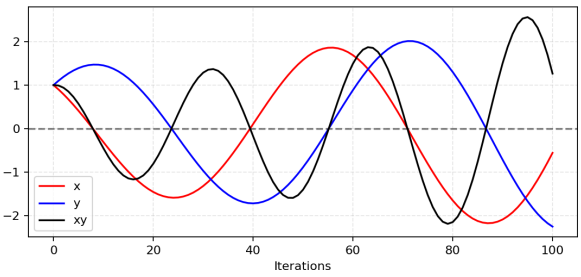


Figure 44: A simulation of our example for updating x to minimize xy and updating y to minimize $-xy$. The learning rate $\eta = 0.1$. With more iterations, the oscillation grows more and more unstable.

5.2 Vanishing Gradient

When the discriminator is perfect, we are guaranteed with $D(x) = 1, \forall x \in p_r$ and $D(x) = 0, \forall x \in p_g$. Therefore the loss function L falls to zero and we end up with no gradient to update the loss during learning iterations. The figure below demonstrates an experiment when the discriminator gets better, the gradient vanishes fast.h

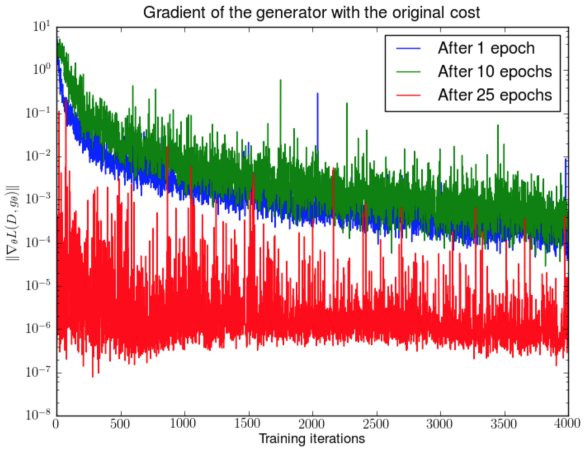


Figure 45: First, a DCGAN is trained for 1, 10 and 25 epochs. Then, with the generator fixed, a discriminator is trained from scratch and measure the gradients with the original cost function. We see the gradient norms decay quickly (in log scale), in the best case 5 orders of magnitude after 4000 discriminator iterations. (Image source: Arjovsky and Bottou, 2017)

As a result, training a GAN faces a dilemma:

- If the discriminator behaves badly, the generator does not have accurate feedback and the loss function cannot represent the reality.
- If the discriminator does a great job, the gradient of the loss function drops down to close to zero and the learning becomes super slow or even jammed.

This dilemma clearly is capable to make the GAN training very tough.

5.3 Mode Collapse

During the training, the generator may collapse to a setting where it always produces same outputs. This is a common failure case for GANs, commonly referred to as Mode Collapse. Even though the generator might be able to trick the corresponding discriminator, it fails to learn to represent the complex real-world data distribution and gets stuck in a small space with extremely low variety.

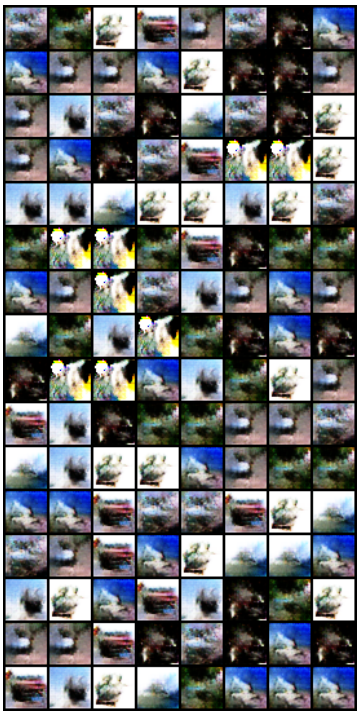


Figure 46: Example of mode collapse: During one of our training of CGAN on CIFAR10, the generated images collapsed to small number of modes, around 6

6 Concluding Remarks

In this project, we've really explored the GANs to great details: not only empirically but also theoretically. This project had definitely been extremely hard for us, due to the scope of the work. And for the past month, we have spent more than 200 hours on training, testing, tuning, and reading the papers and understand about the details of the GAN.

We find this an interesting and rewarding experience, though, as it really teaches us something interesting about doing research in computer-vision and in deep learning-related fields. At last, we'd like to thank each other for our persistent effort and providing support for each other. This project helped us build a strong sense of camaraderie, and helped us get to know each other as well.

References

[1] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio. **Generative Adversarial Network**, *University of Montreal.* `https://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf`

[2] Alec Radford, Luke Metz, **Unsupervised Representation Learning With Deep Convolutional Generative Adaptive Learning** `https://arxiv.org/pdf/1511.06434.pdf`

[3] Martin Arjovsky, Soumith Chintala, Léon Bottou, **Wasserstein GAN** `https://arxiv.org/abs/1701.07875`

[4] **Conditional Generative Adversarial Nets** `https://machinelearningmastery.com/impressive-applications-of-generative-adversarial-networks/`

[5] **18 Impressive Applications of Generative Adversarial Networks (GANs)** `https://arxiv.org/abs/1411.1784`

[6] Tim Salimans et. al. **Improved Techniques for Training GANs** `https://proceedings.neurips.cc/paper/2016/file/8a3363abe792db2d8761d6403605aeb7-Paper.pdf`

[7] Sik-Ho Tsang, **CGAN — Conditional GAN (GAN)** https://medium.com/ai-in-plain-english/review-cgan-conditional-gan-gan-78dd42eee41

[8] Lilian Weng **From GAN to WGAN** https://lilianweng.github.io/lil-log/2017/08/20/from-GAN-to-WGAN.html#low-dimensional-supports

[9] Martin Arjovsky, Leon Bottou **TOWARDS PRINCIPLED METHODS FOR TRAINING GENERATIVE ADVERSARIAL NETWORKS** https://arxiv.org/pdf/1701.04862.pdf

[10] Jason Brownlee, **How to Implement the Inception Score (IS) for Evaluating GANs** https://machinelearningmastery.com/how-to-implement-the-inception-score-from-scratch-for-evaluating-generated-images/

[11] Alec Radford & Luke Metz, Soumith Chintala, **UNSUPERVISED REPRESENTATION LEARNING WITH DEEP CONVOLUTIONAL GENERATIVE ADVERSARIAL NETWORKS** https://arxiv.org/pdf/1511.06434.pdf

[12] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, Li Fei-Fei, **ImageNet Large Scale Visual Recognition Challenge** https://arxiv.org/pdf/1409.0575.pdf

[13] **pytorch-mnist-GAN** https://github.com/lyeoni/pytorch-mnist-GAN

[14] Nathan Inkawhich, **DCGAN TUTORIAL** https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html