Generate Handwritten Digits by GANs

Final Project for ECE-GY 9123 Deep Learning, NYU

Fu-Hsuan Liu
Department of Electrical and Computer Engineering
New York University
fl1392@nyu.edu

May 18, 2021

1 Introduction

This project utilizes two types of Generative Adversarial Network(GAN)-based techniques, Vanilla GAN [1] and Deep Convolutional GAN (DCGAN) [2], to generate handwritten digits images.

First we will introduce the model structures and conduct the experiments. Then we will analyze the results discussing how different parameters settings will affect the performance of GANs. Finally, compare the results from both versions of GANs in various aspects. The implementation of this project can be found at this *repository*.

2 Models

The GAN models consists of two neural networks: *Generator*(G) and *Discriminator*(D). The role of G is to take noises (z) as input and produce fake samples $X_fake = G(z)$, while the role of D is to take datapoints as input and distinguish whether they're real or fake: $D(G(z))$ and $D(X_real)$.

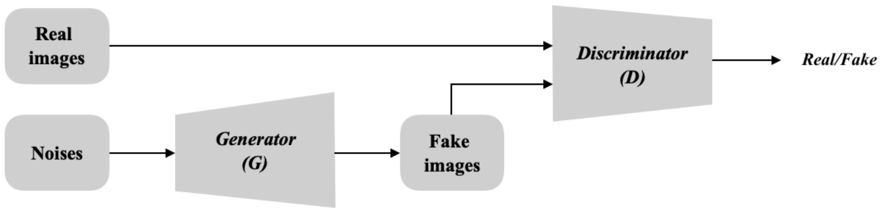


Figure 1: General architecture of GAN-based models

Our goal is to use fake samples generated by G as the input of the D, and fool D, making it recoginize them as real. Therefore, G and D are opposite to each other in training. In optimization strategy, it is a two-player turn-based $minmax$ game, where min means we need to minimize G loss and max means to maximize D's loss.

$$\mathcal{L} = min\ max(L_D, L_G)$$

$$\mathcal{L}_D = logD(X_{real}) + log(1 - D(G(z)))$$

$$\mathcal{L}_G = log(1 - D(G(z)))$$

Both Vanilla GAN and DCGAN follow this core idea in training. The main difference for two versions of GANs is the architecture of G and D.

2.1 Vanilla GAN

Vanilla GAN is one of the simplest types of GAN. We follow the implementation policy in [1]. G takes a noise with dimension = 100. After flattening data, pass though fully-connected layers, followed by LeakyReLU and Dropout. Please note that we choose LeakyReLU over traditional Relu is because of LeakyReLU does not have zero slope and it's more balanced, so it will not have dead neurons and we can care less about the initialization. The (fully-connected - LeakyReLU - Dropout) stack repeat three times, and then wrapping up by a tanh. Finally, output data with shape $(28 \times 28 \times 1)$.

The structure of D is similar to G, but the dimension of output after each layer is in a opposite direction to D, and end up with a sigmoid activation function.

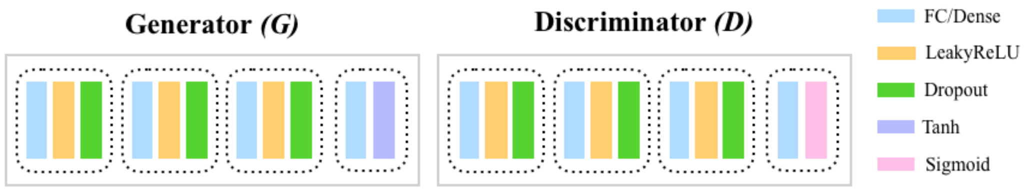


Figure 2: Generator and Discriminator structure in Vanilla GAN.

Please see **Apendix A**, table 1 and table 2 for more Vanilla GAN architecture and hyperparameters details.

2.2 DCGAN

DCGAN removes fully connected layers used in Vanilla GAN, and replaces it with convolutional stride. In addition, a batch normalization layer is added before LeakyReLU, which helps accelerate the training [3].

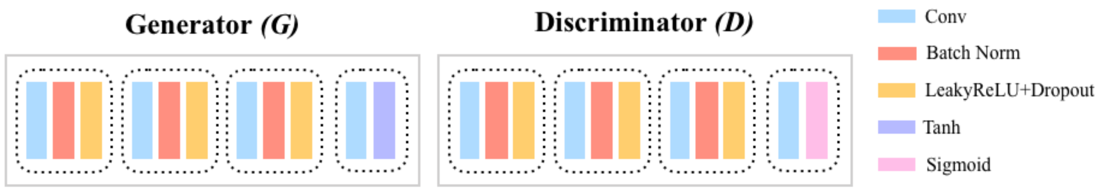


Figure 3: Generator and Discriminator structure in Vanilla GAN.

Please see **Apendix A**, table 3 and table 4 for more DCGAN architecture and hyperparameters details.

3 Experiment

In this section, we will focus on the experiment settings details including environment setup, training parameters for our GAN-based models.

3.1 Environment

The experiments are running under Tesla T4 GPU with NVIDIA-SMI CUDA version 11.2, Intel (R) Xeon(R) @ 2.20GHz CPU model. We choose TensorFlow keras for the implementation of GAN models as well as optimizer and loss function. Other open-source pakages such as NumPy, Matplotlib, time, os, imageio are used for foundamental operations or data/result visualization.

3.2 Datasets

To generate handwritten digits, we choose MNIST dataset [4], which is widely used in the field of machine learning from earlier times. MNIST is a large dataset of grayscale handwritten digits samples, including 60000 training samples and 10000 testing samples, and each sample is a 28×28 pixel image. We normalize the image and makes each picel value between [-1, 1] before training.

3.3 Training, Optimizer, Loss fuction

For vanilla GAN and DCGAN, both their Loss criterion is Cross-Entropy loss function, and the optimizer is Adam algorithm according to the implementation in [1] and [2].

The optimizer uses learning rate `lr = 0.0002` and `betas = (0.5, 0.999)` in our experiment. Note that though `lr = 0.0001` and `betas = (0.9, 0.999)` were suggested in paper [1], later research such as [2], mostly adjust lr to 0.0002 and $betas$ to (0.5, 0.999) for more stable training.

Other training settings are `EPOCHS = 40`, and `batch_size = 64` and `samples = 100 (10*10)`. During each epoch, dataset are divided into batches, D and G are trained based on their loss, and do the gradient descent/acsent. In the end, a fixed noise seed will apply to our trained G.

4 Results

From figure 5, that G and D are competitive to each other, if one increases, the other one decreases. G loss starts reaching minimum at aruond epoch 20, which corresponds to the epochs we need to generate fake images from vague to clear and fool D. Interestingly, there are more '0' and '7' digits among the samples in DCGAN, it may becuase such digits have not too complex shape, so they can easily deceive D.

More generated images results with sample = 16 versus epochs under different batch sizes can be found at **Appendix B**, and their loss plots can be found at **Appendix C**.

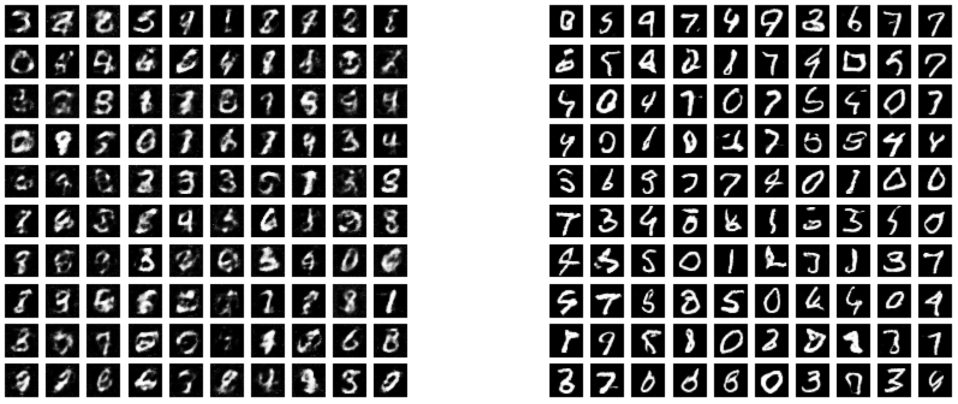


Figure 4: G-generated images by Vanilla GAN (left) and DCGAN (right) at epoch 40.

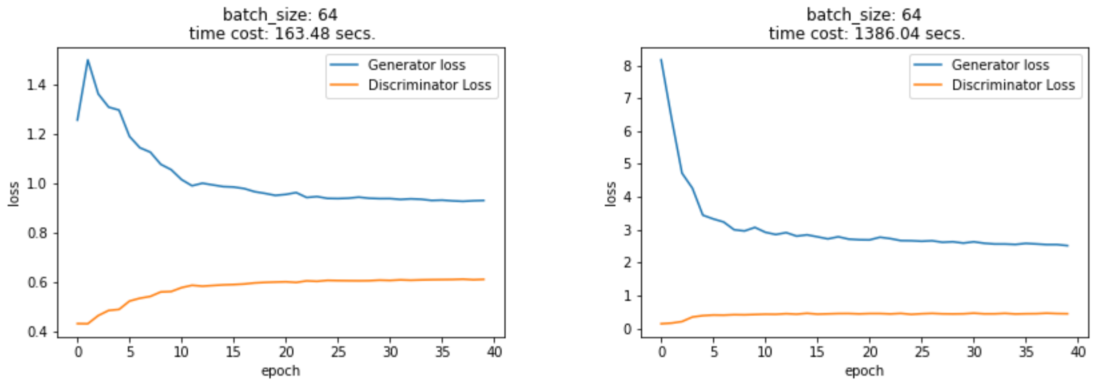


Figure 5: Loss and time cost for Vanilla GAN (left) and DCGAN (right).

5 Discussion

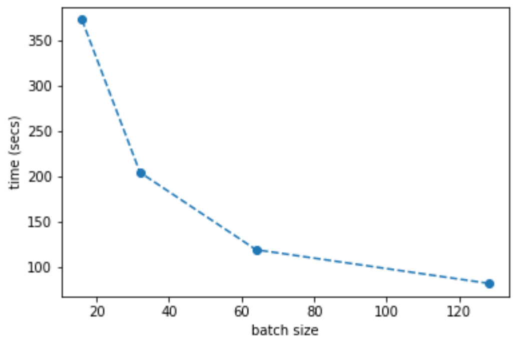
This section shows how different parameter settings affect the performance. In order to analyze their relation more precisely and fairly, we only change one parameter while remain other hyper parameters unchanged at a time. In addition, a random seed is set to ensure the fixed noise did not make too much uncertainty.

Kernel size

In DCGAN, the smaller the kernel size, the longer the training time. It also shows higher G loss in comparison to larger kernel size from the start to the end.

Batch size

It's intuitively that training time is inversely proportional to batch size, since larger batch leads to less computation in each training epoch. From our GAN results, we can see that, to be more specific, the training time is inversely exponential to the batch size. Surprisingly, there is not much difference for loss value in Vanilla GAN. However, though the plot below shows 4 lines of discriminator loss are very similar, the number of epoch to reach equilibrium are all around $20 \sim 30$, we can easily find out the early stopping point, or the optimal epoch number which the generator reached the



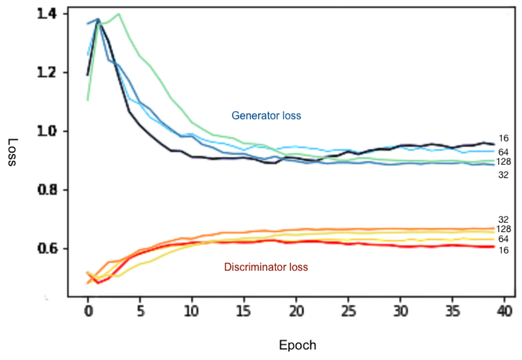


Figure 6: (Left) Vanilla GAN training time under different batch size

Figure 7: (Right) Vanilla GAN training G and D loss under different batch size

minimum steadily, is that larger batch size has larger epoch number, and vice versa. It's obvious when $bs = 128, epoch \approx 30$, while $bs = 16,\ epoch \approx 20$.

Gan types

From the results of Vanilla GAN and DCGAN, We can see the trade-off between image quality and training time. DCGAN outputs better quality images since the edges of the generated digits are more clear, precise, and less noises. However, it also takes longer time. From figure 5, one epoch takes an average of 4.075 seconds for basic GAN while 34.6 seconds for DCGAN. From Appendix A table 1 to 4, we can see the total number of parameters of DCGAN is around 2.5 times Vanilla GAN's, but the time costing is way longer than vanilla GAN due to the architecture complexity.

6 Conclusion

This project introduces two types of GAN-based models. We successfully implement the architectures and generate fake handwritten digits images with rational time and accuracy. The results shows the images generated by DCGAN are clearer, less blurriness and noises, which meets our expectation. There is still a big room for improvement. Now, not every digit is generated equally in the samples. Different versions of GANs such as conditional GAN (cGAN) is designed for addressing this issue by training with labels.

References

[1] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. *Generative Adversarial Networks*. 2014.

[2] Alec Radford, Luke Metz, and Soumith Chintala. *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*. 2016.

[3] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015.

[4] Y. LECUN. The mnist database of handwritten digits. *http://yann.lecun.com/exdb/mnist/*.

Appendix

A Model Implementation details

A.1 Vanilla GAN

Layer (type)	Output Shape	Param #
Flatten	(784)	0
Dense	(512)	401920
LeakyReLU	(512)	0
Dropout	(512)	0
Dense	(256)	131328
LeakyReLU	(256)	0
Dropout	(256)	0
Dense	(128)	32896
LeakyReLU	(128)	0
Dense (sigmoid)	(1)	129
Total params: 566,273		
Trainable params: 566,273		
Non-trainable params: 0		

Table 1: *Vanilla GAN* Discriminator

Layer (type)	Output Shape	Param #
Dense	(128)	12928
LeakyReLU	(128)	0
Dropout	(128)	0
Dense	(256)	33024
LeakyReLU	(256)	0
Dropout	(256)	0
Dense	(512)	131584
LeakyReLU	(512)	0
Dropout	(512)	0
Dense (tanh)	(784)	402192
Reshape	(28, 28, 1)	0
Total params: 579,728		
Trainable params: 579,728		
Non-trainable params: 0		

Table 2: *Vanilla GAN* Generator

A.2 DCGAN

Layer (type)	Output Shape	Param #
Conv2D	(14, 14, 64)	1664
BatchNorm	(14, 14, 64)	256
LeakyReLU	(14, 14, 64)	0
Dropout	(14, 14, 64)	0
Conv2D	(7, 7, 128)	204928
BatchNorm	(7, 7, 128)	512
LeakyReLU	(7, 7, 128)	0
Dropout	(7, 7, 128)	0
Flatten	(6272)	0
Dense (sigmoid)	(1)	6273
Total params: 213,633		
Trainable params: 213,249		
Non-trainable params: 384		

Table 3: *DCGAN* Discriminator

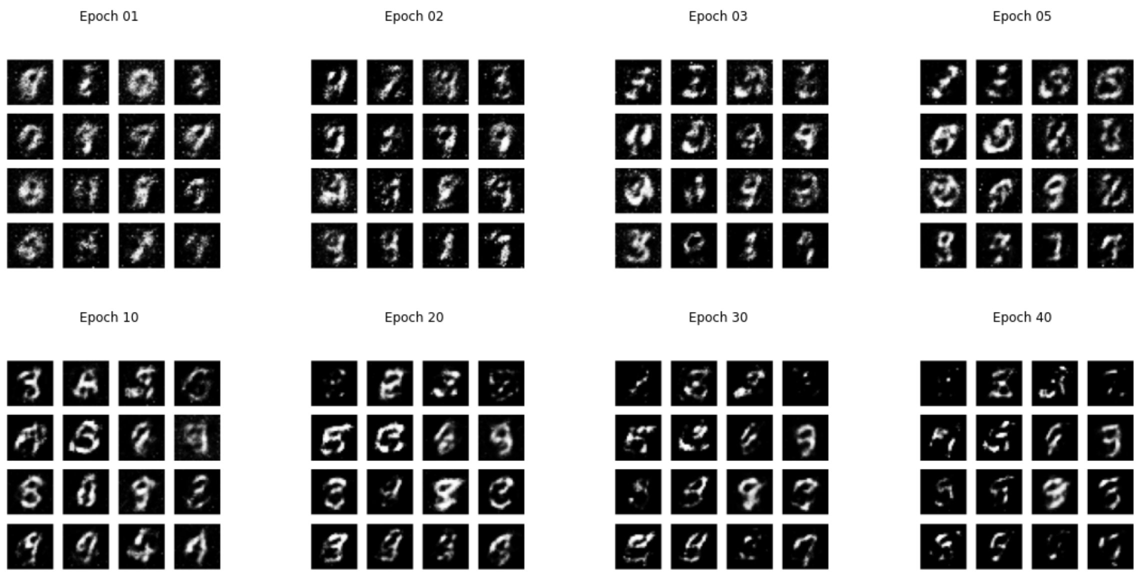
Layer (type)	Output Shape	Param #
Dense	(12544)	1254400
BatchNorm	(12544)	50176
LeakyReLU	(12544)	0
Reshape	(7, 7, 256)	0
Conv2DTrans	(7, 7, 128)	819200
BatchNorm	(7, 7, 128)	512
LeakyReLU	(7, 7, 128)	0
Conv2DTrans	(14, 14, 64)	204800
BatchNorm	(14, 14, 64)	0
Conv2DTrans (tanh)	(28, 28, 1)	1600
Total params: 2,330,944		
Trainable params: 2,305,472		
Non-trainable params: 25,472		

Table 4: *DCGAN* Generator

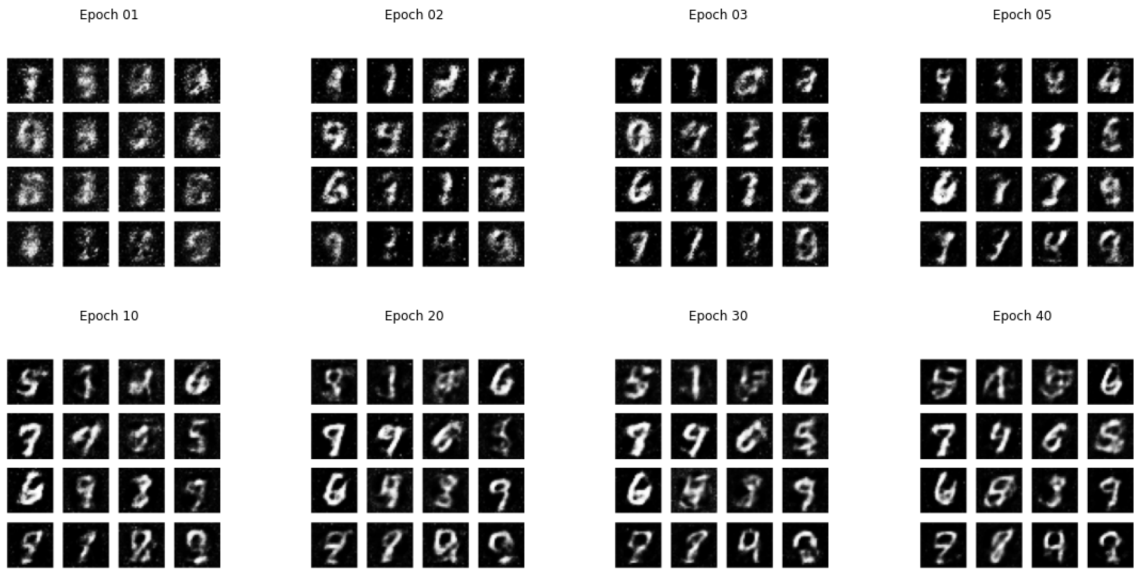
B Generated images in different batch sizes

To observe the quality change of the fake images generated by G during the training process, select specific epochs: 1, 2, 3, 5, 10, 20, 30, 40.

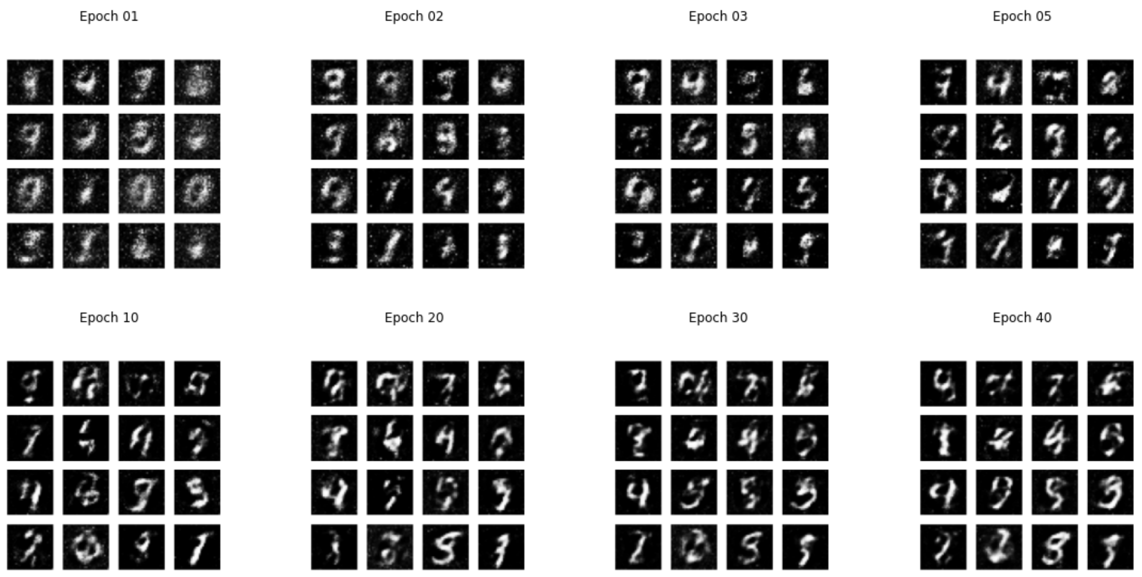
B.1 Vanilla GAN



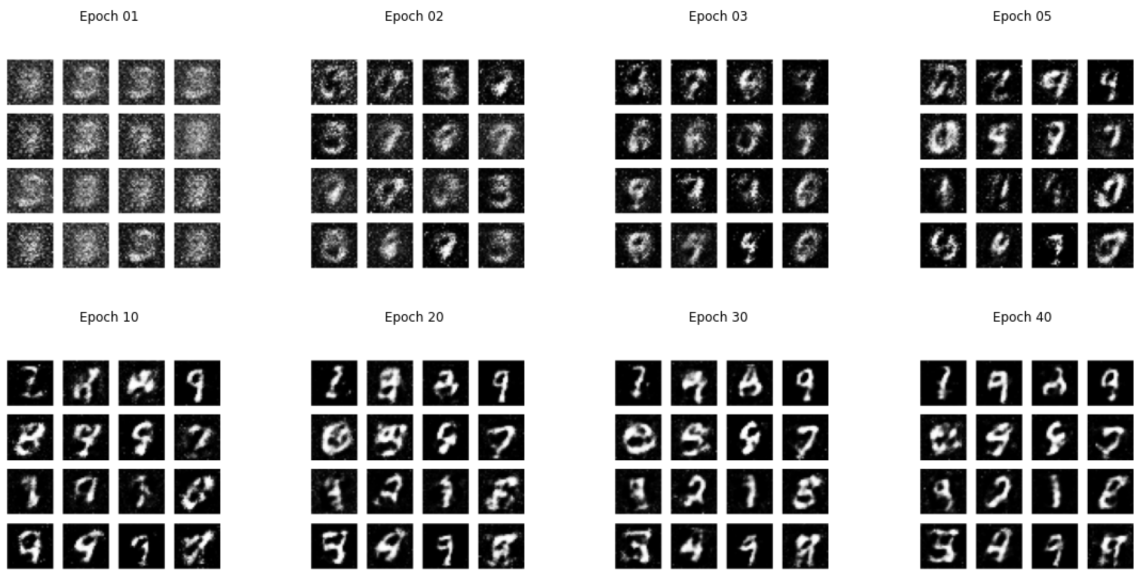
`bs = 16`



`bs = 32`

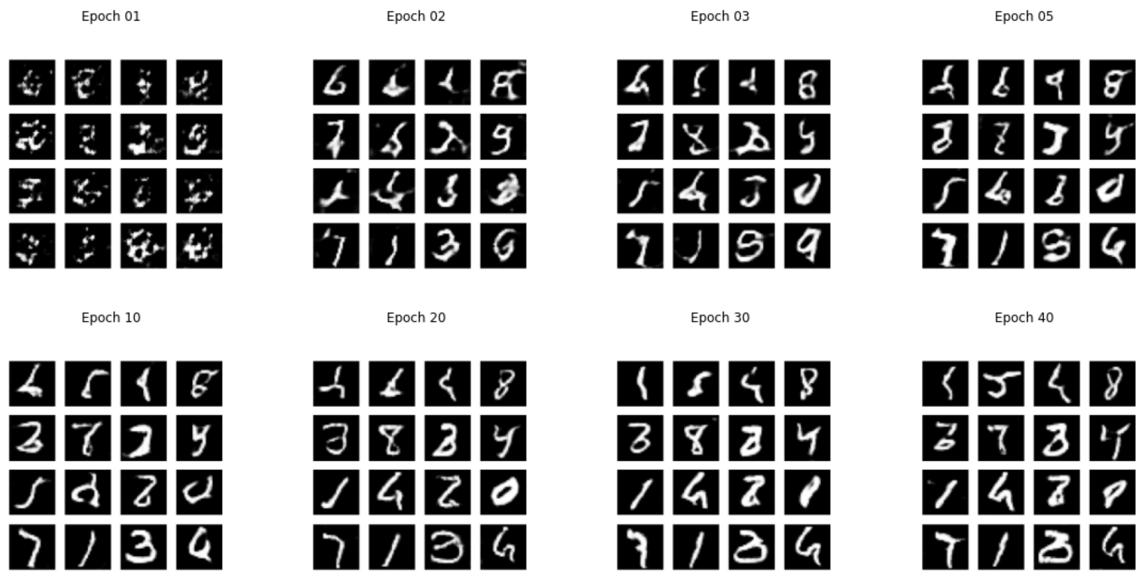

Epoch 01
Epoch 02
Epoch 03
Epoch 05
Epoch 10
Epoch 20
Epoch 30
Epoch 40

bs = 64

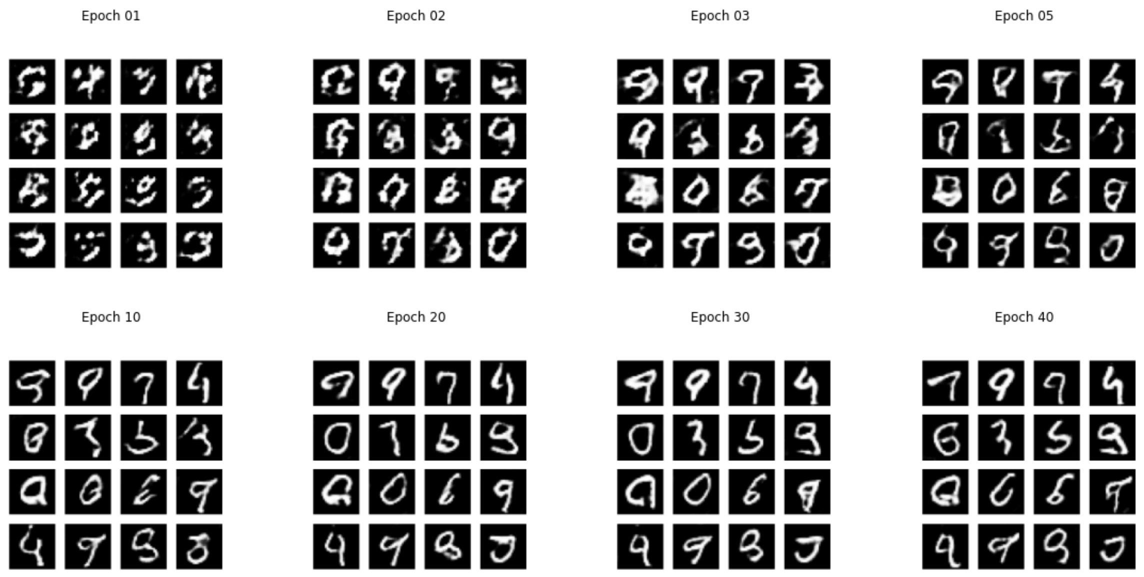

Epoch 01
Epoch 02
Epoch 03
Epoch 05
Epoch 10
Epoch 20
Epoch 30
Epoch 40

bs = 128

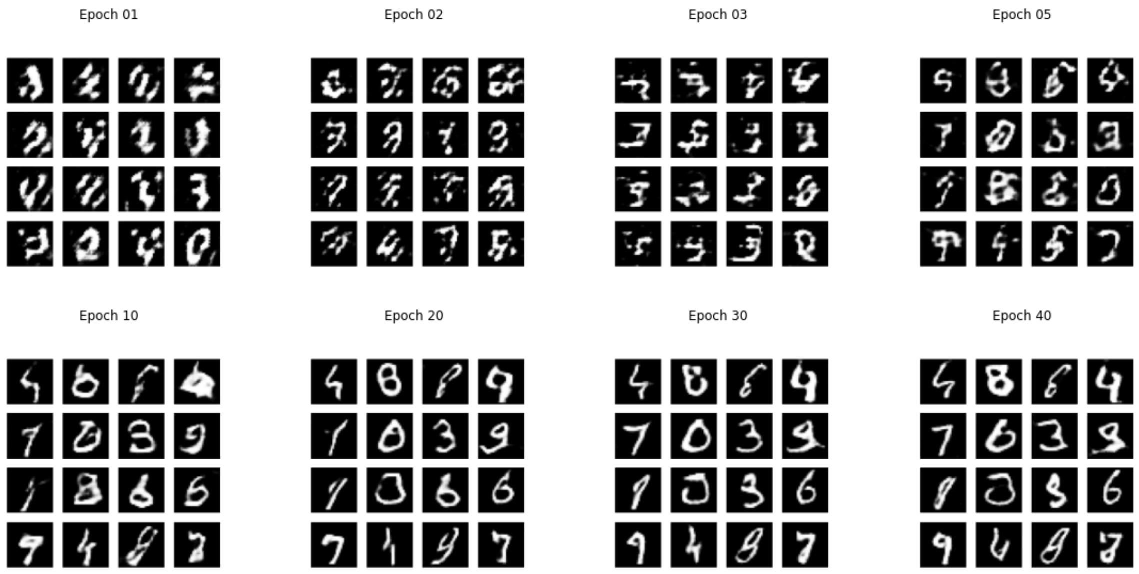
B.2 DCGAN



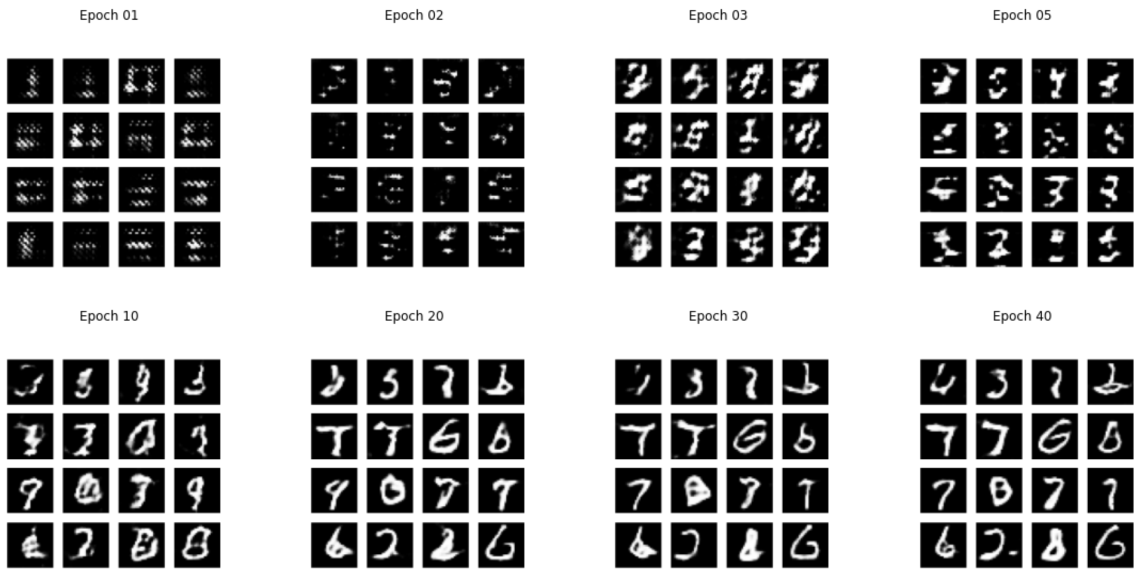
bs = 16



bs = 32


Epoch 01
Epoch 02
Epoch 03
Epoch 05
Epoch 10
Epoch 20
Epoch 30
Epoch 40

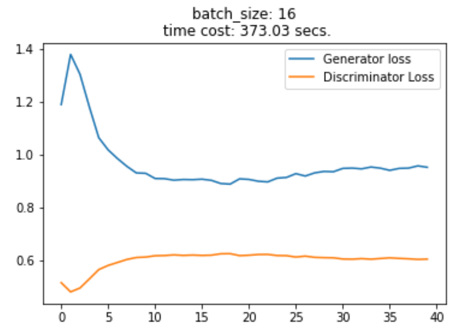
bs = 64

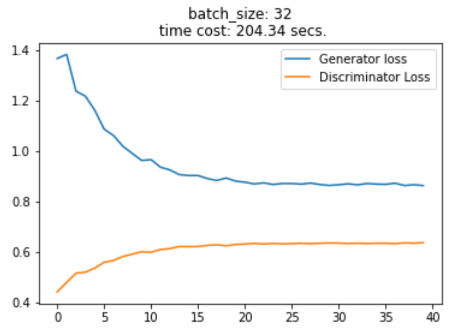

Epoch 01
Epoch 02
Epoch 03
Epoch 05
Epoch 10
Epoch 20
Epoch 30
Epoch 40

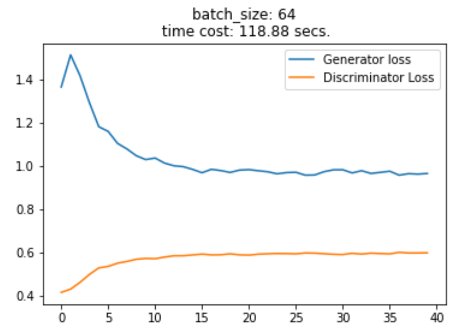
bs = 128

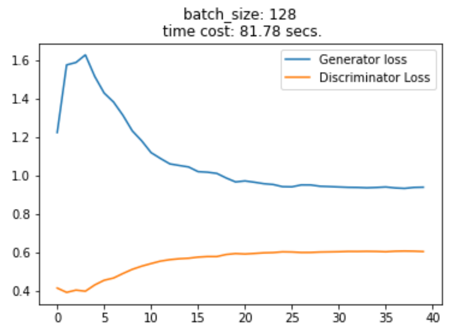
C Loss and training time in different batch sizes

C.1 Vanilla GAN

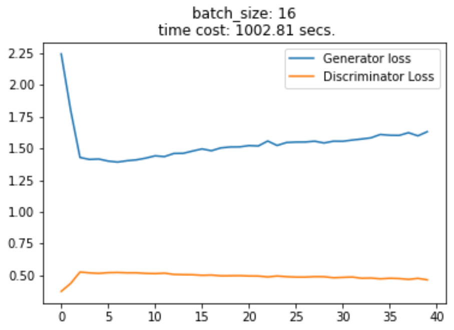


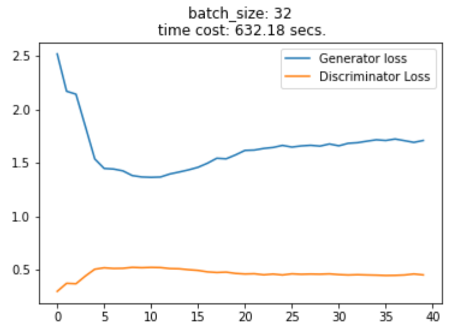


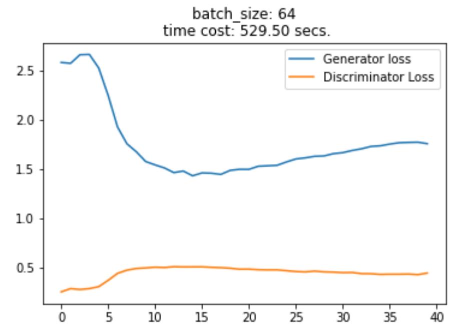


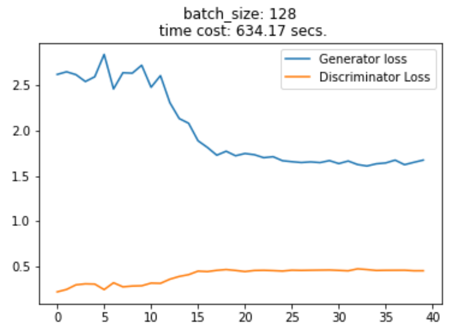


C.2 DCGAN


batch_size: 16
time cost: 1002.81 secs.
Generator loss
Discriminator Loss


batch_size: 32
time cost: 632.18 secs.
Generator loss
Discriminator Loss


batch_size: 64
time cost: 529.50 secs.
Generator loss
Discriminator Loss


batch_size: 128
time cost: 634.17 secs.
Generator loss
Discriminator Loss