Weekly Report(Aug 2nd - Aug 8th, 2019)

Jianghao Lin
Shanghai Jiao Tong University
`chiangel.ljh@gmail.com`

Abstract

I read the paper *A Style-Based Generator Architecture for Generative Adversarial Networks* published by NVIDIA in 2019, make a summary of basic ideas of other GAN models by going through papers and blogs. Also, I continue my trying on implementation and practice in GAN models.

1 Style GAN

1.1 Mapping Network

In traditional GAN models, we will feed a latent vector into the generator and get an output image. We expect that the value of each dimension of the latent vector can respectively control some style of the generated image (e.g. hair color, wearing a glass or not). But the reality is not that satisfying. Take the following situations as examples:

- We want to modify the hair color of our generated face by nudging a number in the latent vector. But the output does not only change the hair color, but also change the skin color or even gender.
- We want to modify the hair color of our generated face. But we have to modify several numbers of the latent vector instead of one. Also this can lead to other unexpected changes on the generated image.

The above situation is called **feature entanglement**. To solve this problem, StyleGAN introduces a network that maps an input vector to a intermediate latent vector which the generator uses. This new and relatively small network is named **mapping network**. The architecture of mapping network is shown in Figure 1.

Adding this neural network to create an intermediate latent vector would allow the GAN to figure out how it wants to use the numbers in the vector we feed in. Therefore the problem of feature entanglement is alleviated.

1.2 Adaptive Instance Normalization (AdaIN)

Suppose we got a latent vector with very low feature entanglement. Maybe the first number of the vector indicates the hair color and the second number determines the gender etc.

Now think about how do we draw a picture from such description of latent vectors. We do not look at the description once and paint the image immediately. More likely we will first paint the hair, look at another description detail again and then paint another style.

So, instead of feed the latent vector to the generator once for all, StyleGAN feeds the latent code to every layer of the generator independently.

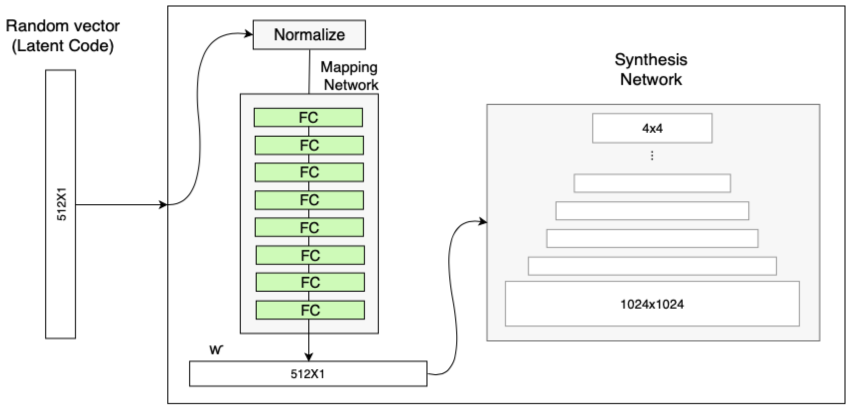


Figure 1: Mapping Network

In order to apply this idea, StyleGAN uses **Adaptive Instance Normalization**. AdaIN is the learned affine transformation (actually equivalent to linear layer).

$$y = (y_s, y_b) = f(w) haiyou \tag{1}$$

$$AdaIN(x_i, y) = y_s \frac{x_i - \mu(x)}{\sigma(x)} + y_b \tag{2}$$

Here, x_i is the input of every layer in the generator and y comes from the latent vector.

1.3 Constant Input

Once again, let us think about drawing a ground true picture by hand. All images from the same kind will have a roughly similar skeleton and we just add and modify several details and styles. But in traditional GAN, the only source of variation and stylistic data is the input latent vector so we have to keep it varying.

However, in StyleGAN, the style and details can be controlled by mapping network and AdaIN, so it is alright to keep the input of generator a constant tensor to indicate the skeleton of an image. Note that it is the input of generator that is a constant - not the input of mapping network. Thus we have the basic architecture of StyleGAN in Figure 2.

1.4 Style Mixing

Because the latent vector is injected into every layer independently, so we can inject several different latent vector in different layers in order to get a mixture feature style of them.

In the experiment that NVIDIA makes, 3 different vectors were injected into 3 different spots:

1. At the coarse layers, where layers where the hidden representation is spatially small from 4×4 to 8×8.
2. At the medium layers, where layers where the hidden representation is medium sized from 16×16 to 32×32.
3. At the fine layers, where layers where the hidden representation is spatially large from 64×64 to 1024×1024.

The spots seems to be set in a not-even way, but it is reasonable because the generator quickly picks up on information and the larger layers mostly refine and sharpen the outputs from the previous layers.

As a result, the mixture output will combine the feature of all three original images (e.g. the mixture faked face may have the same hair color as original image 1 but the same skin color as image 2).

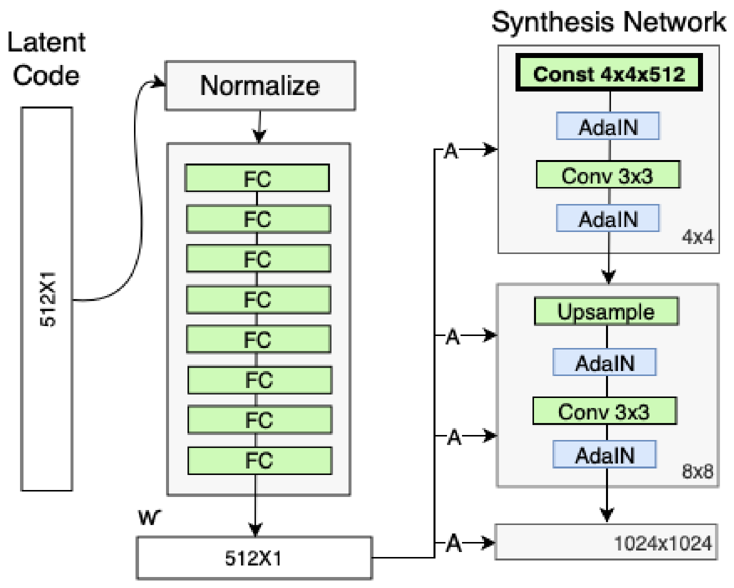


Figure 2: Basic Architecture of StyleGAN

1.5 Stochastic Noise

In order to make the generated image more convincing, StyleGAN introduces the **Stochastic Noise**, which contribute to slightly different hairstyles or freckles. Because we keep the input of generator a constant, so we all apply layer-wise noise injection rather than modifying the latent vector.

The final architecture of StyleGAN is shown in Figure 3.

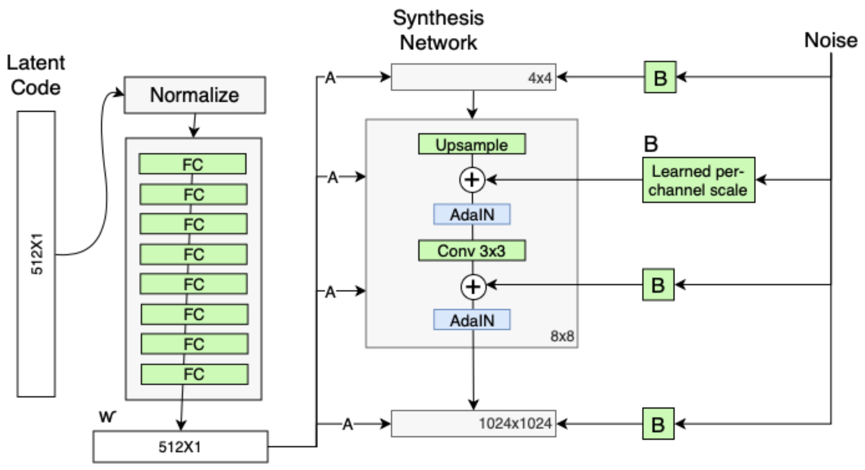


Figure 3: Final Architecture of StyleGAN

2 Summary of GAN zoos

This part I summarize the basic idea of some GANs while I go through the papers blogs and documents. Some details may be missing because I do not have time to read them line by line.

2.1 DCGAN: Deep Convolutional GAN

DCGAN apply the transposed convolutions to the generator and discriminator network instead of the simple linear layers in the original GAN. It improves the quality of generated images.

2.2 CGAN: Conditional GAN

In traditional GAN, we can only generate images from a random noise vector z, but can not control the exact kind of the output - we are not sure the output will be a car, a trunk or a train. By adding auxiliary information (e.g. class type, texture description or voices) to **both the generator and discriminator network**, CGAN can control the output to be what we want it to be.

As for how to combine the auxiliary information, simply concating two tensors is alright. Also, if the information is the class type of the image, use one-hot vector code to encode it.

2.3 CycleGAN

CycleGAN provides an unsupervised solution for unpaired image to image translation. CycleGAN operates on two domain X and Y, consisting of two generators $G_{X\to Y}$ and $F_{Y\to X}$, and two discriminators D_X and D_Y. The core idea is **cycle consistency**:

$$F(G(x)) \approx x, x \in X \tag{3}$$

$$G(F(y)) \approx y, y \in Y \tag{4}$$

Thus we will add a cycle consistency loss to the original loss of generator:

$$L_{GAN}(G, D_Y, X, Y) = E_y[log(D_Y(y))] + E_x[log(1 - D_Y(G(x))] \tag{5}$$

$$L_{cycle} = E_x[||F(G(x)) - x||_1] + E_y[||G(F(y)) - y||_1] \tag{6}$$

$$Loss = L_{GAN} + L_{cycle} \tag{7}$$

2.4 CoGAN: Coupled GAN

CoGAN provides an unsupervised solution to learn joint distribution of data from different domains and it consists of two generators and two discriminators. But it have an assumption that domain X and domain Y have highly similar semantic similarity.

Based on this assumption, CoGAN learned joint distribution by enforcing weight sharing constraint on its high level representation weights and then sharpen each domain's detail respectively. Thus the output of G_1 and G_2 will have some kind of corresponding relations.

The core idea of CoGAN is shown in Figure 4.

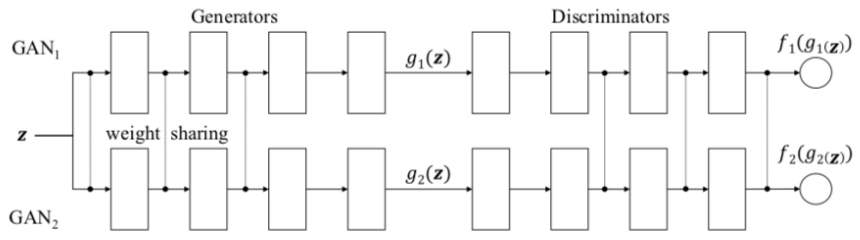


Figure 4: CoGAN

2.5 ProGAN: Progressive Growing of GAN

It is painful to train a GAN model due to the training instability - sometimes the loss of the GAN oscillates or even explode. ProGAN is a technique that helps stabilize GAN training by incrementally increasing the resolution of the generated image.

Instead of directly generate a 1024×1024 image, we will start from a 4×4 image and gradually increase the image resolution to 8×8, 16×16 and finally 1024×1024. Also when feed a 1024×1024 image to the discriminator, we also apply such a degressively process from 1024×1024 to 512×512, 256×256 and finally a scalar.

The architecture of ProGAN is shown in Figure 5.

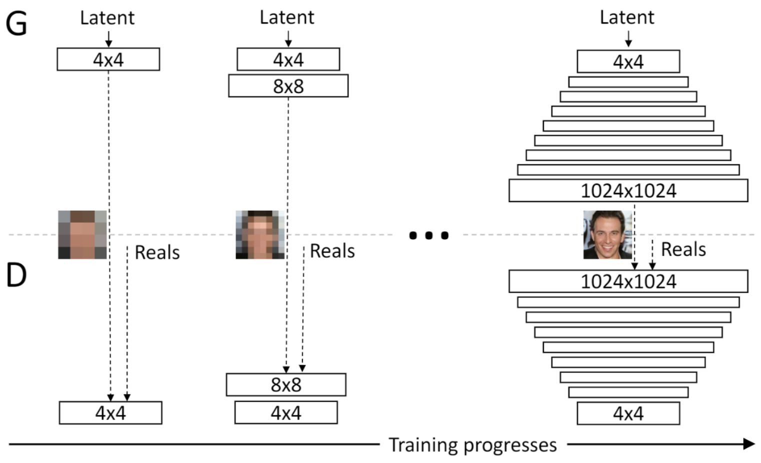


Figure 5: ProGAN

2.6 StarGAN

In area of image to image translation, most models like CycleGAN can only tranform from one domain to another. But what if we want to transform in multiple domains? We should train $k \times (k-1)$ GAN models, which is definitely impracticable. StarGAN comes up with an idea to solve this problem using only one generator.

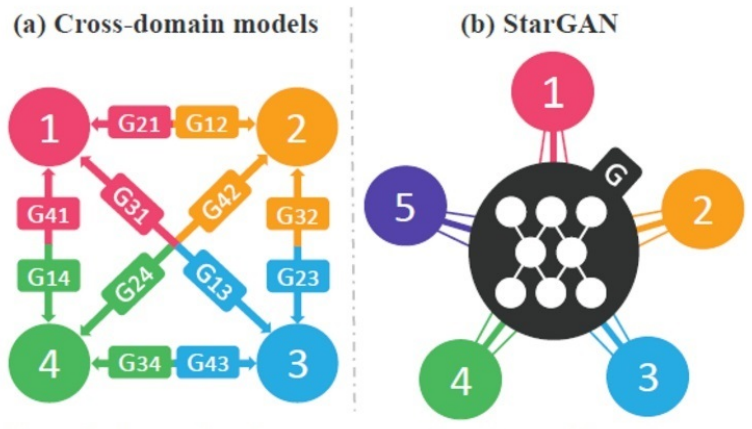


Figure 6: StarGAN

The generator takes an image and the target domain as inputs and generates the corresponding fake image. The discriminator takes an image as an input and output a scalar indicating whether it is real or not and a domain classification.

The core of training is similar to cycle consistency. When we re-generate an image from the target domain back to the original domain, the output should be as close as the original input image. The architecture of StarGAN is shown in Figure 7.

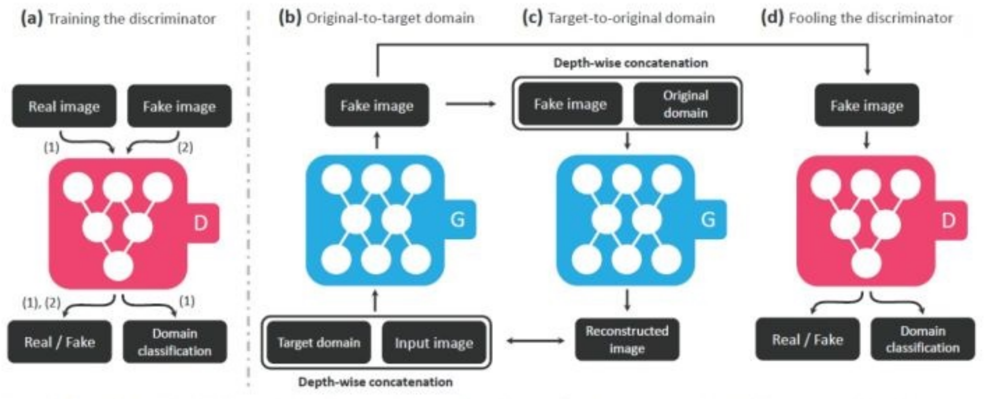


Figure 7: Architecture of StarGAN

3 Implementation and Practice

3.1 CGAN on MNIST

Last time I mentioned that I failed to build a CGAN based on CONV layers. This might be a way too much step for me. So this time I start from the original CGAN, which is built by pure linear neurons and ReLU activations without convolutional or pooling operations.

The CGAN is built and Figure 8 show the generated images from the initial CGAN after 200-epoch training. But it is obviously dissatisfying. After carefully checking the codes and comparing them to the source code on Github, I find out that I mistakenly pick the MSE Loss instead of BCE Loss! I even do not know how I could make this stupid mistake :(

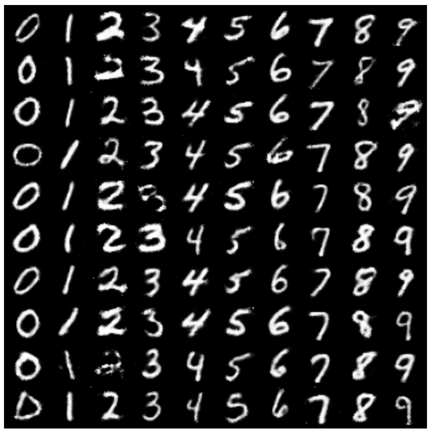
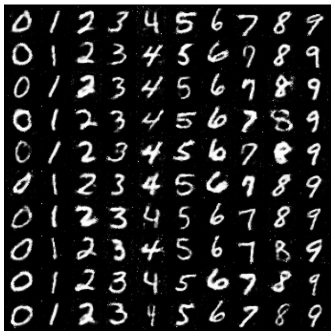
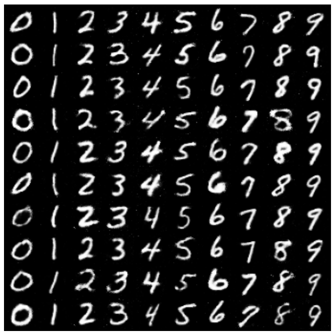
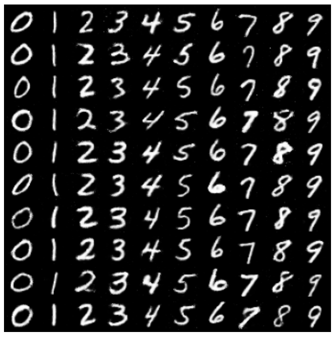


Figure 8: Output of initial CGAN

After correcting the loss mistake, I train the model for 800 epochs. As we can see in Figure 9, the quality of generated images continue growing as the training process moves on. And after 800-epoch training, I think the generated images are rather realistic despite a little pixel blurry.


(a) Epoch 200


(b) Epoch 400


(c) Epoch 600

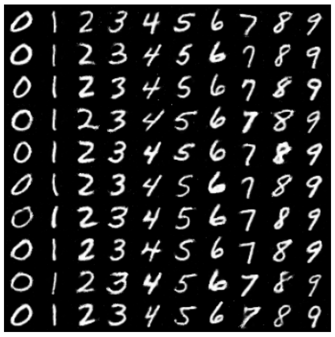

(d) Epoch 800

Figure 9: Output of corrected CGAN

Additionally, as we all known, the latent vector z somehow represents the feature of the generated images. So I think maybe dimension 100 is so large for simple images in MNIST dataset (only hand write numbers from 0 to 9) that it can cause code information redundancy and feature entanglement. Therefore, I try to reduce the dimension of latent vector from 100 to 10, but it turn out a bad result shown in Figure 10.

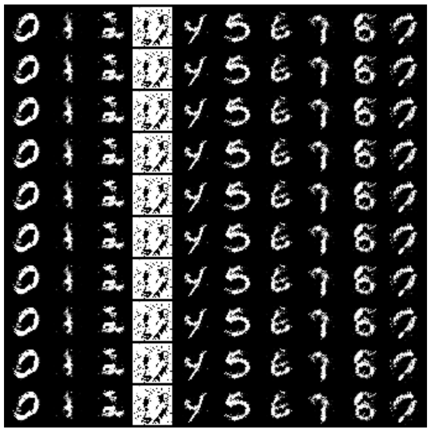


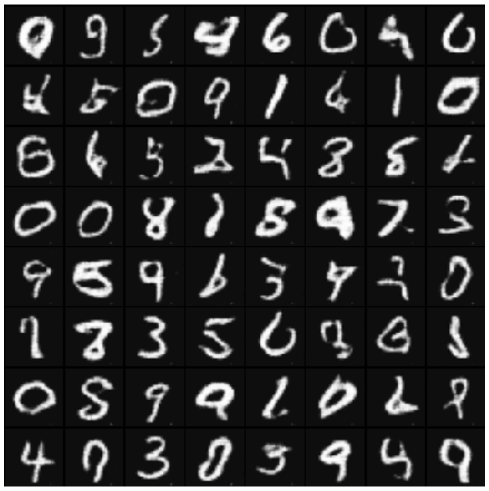
Figure 10: CGAN with latent dimension of 10

In my opinion, it is because the latent space are too small to extract the full information for image generation. This means that a seemingly simple image (e.g. hand-write number 0) can contain many feature information for machines which humans may not notice.

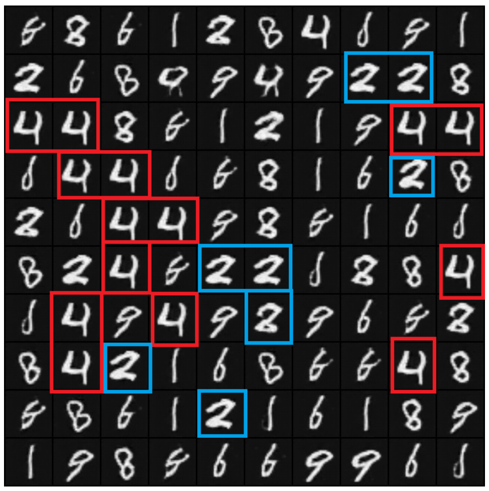
3.2 DCGAN on MNIST

Last time I built a DCGAN model, trained it for 50 epochs and considered it converged. But this time I extend the training epoch to 200 and find out that the work I done last time is far away from accomplishment.

As the two figures shown in Figure 11, after 50-epoch training last time, the generated images still greatly suffer from blurry strokes with white pixel points scattering around, which can be easily judged to be fake one by a human being. However, after 200-epoch training this time, the generated strokes are so clean, clear and sharply that one can not easily say it is faked or not in spite of the semantic interpretation of some weird images.



(a) Output of generator last time

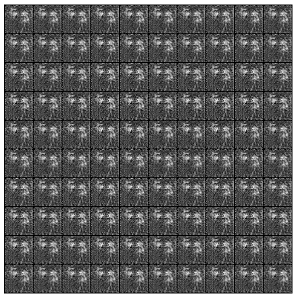


(b) Output of generator this time

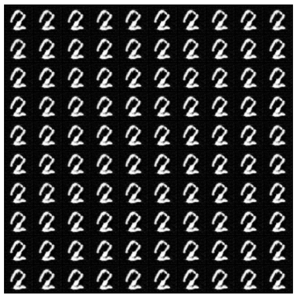
Figure 11: Output of Generator Network

We can see that most of the generated images clearly indicate a corresponding number. Nevertheless, some of them are just uninterpretable strange symbols. I think this is due to the basic idea of GAN. The goal of GAN is to make the distribution p_G exactly the same as p_{data}, but two distribution must have overlaps and differences. The generator may combine the strokes of different numbers and thus create a weird symbol. Therefore, next time I will try to combine the DCGAN with CGAN, which should solve both the problems - semantic interpretation and clearly strokes.

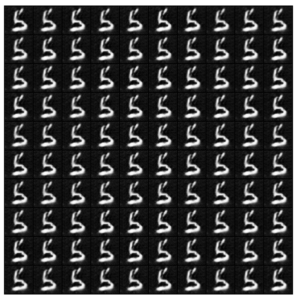
Also, I notice that many generated images are exactly the same (I have labeled some of them using red and blue box selections), which means that mode collapse has **partially** taken place. Actually this bad smell occurs when I print out the intermediate output graphs (shown in Figure 12). Unfortunately, I have not come up with ideas to successfully solve it up to now :(



(a) Epoch 19



(b) Epoch 21


(c) Epoch 23

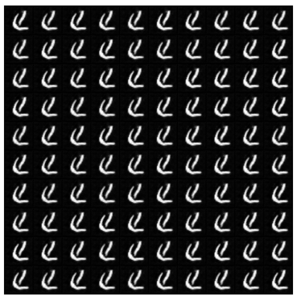

(d) Epoch 25

Figure 12: Intermediate Images of DCGAN

4 Work Next Week

- Paper about BigBiGAN
- Presentation about StyleGAN
- Try to implement more complex GAN model
- Fill in the summary of GAN zoo