Weekly Report(July 12th, 2019)

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

Abstract

This is a collection of my paper notes for future reference. The paper note contain two parts: *Read in details* and *Read briefly*. The former part consists of papers which I read in details, usually containing derivations and interpretations of mathematical formulas. The latter part only contain the core idea of the GAN model in the paper and is usually lack of rigorous demostration.

Contents

1 (In Detail) GAN: Generative Adversarial Network

1.1 Basic idea of GAN

Generative adversarial nets are based on a game theoretic scenario where the generator network directly produces images x from a random noise z and another discriminator network tries to distinguish the generated image from the real ones.

Ideally, the distribution of the generated data will be the same as the real data image and the accuracy of the discriminator network will be 0.5 because it can not tell an image is actually fake or not.

1.2 How to measure the divergence ?

1.2.1 KL divergence

KL divergence is also called relative entropy.

For discrete probability distributions P and Q defined on the same probability space, the KL divergence between P and Q is defined to be:

$$KL(P||Q) = \sum_{x} P(x)log(\frac{P(x)}{Q(x)}) \tag{1}$$

For continuous random variables, it is defined to be:

$$KL(P||Q) = \int_{-\infty}^{\infty} p(x)log(\frac{p(x)}{q(x)})dx \tag{2}$$

We can have the inequation because the log function is a convex function:

$$\begin{aligned} KL(P||Q) &= \int_{-\infty}^{\infty} p(x)log(\frac{p(x)}{q(x)})dx \\ &= -E(log\frac{P(x)}{Q(x)}) \\ &\geq -log(E(\frac{P(x)}{Q(x)})) \\ &= -log(P(x) \times \frac{P(x)}{Q(x)}) = 0 \end{aligned} \tag{3}$$

Therefore, KL divergence is in range of $[0, 1]$. KL divergence is equal to 0 if and only if two distribution is exactly the same. And the smaller the KL divergence is, the more similar two distributions are.

1.2.2 JS divergence

It is obvious that KL divergence is asymmetric, which means that $KL(P||Q) \neq KL(Q||P)$. So we introduce the JS divergence:

$$JS(P||Q) = \frac{1}{2}KL(P||M) + \frac{1}{2}KL(Q||M) \tag{4}$$

where $M = \frac{1}{2}(P + Q)$. The range of JS divergence is $[0, log2]$. JS divergence is obviously symmetric and the smaller the JS divergence is, the more similar two distributions are.

1.3 Definitions in GANs

- **Generator G** - G is a function that takes a vector z and outputs x. So given a prior distribution $P_z(z)$, then a probability distribution $P_G(x)$ is defined by function G.
- **Discriminator D** - D is a function (actually a binary classification) takes x as inputs and outputs a scalar in range of $[0, 1]$ to tell the input x is faked or not.
- **V(G, D)** - The function V(G, D) is defined to be:

  $$V(G, D) = E_{x\sim p_{data}}[log(D(x))] + E_{z\sim p_z}[log(1 - D(G(z)))] \tag{5}$$

  Defining V(G, D) like this allows the following missions:

  1. Given a G, $\max_D V(G, D)$ evaluates the difference between p_G and p_{data}.
  2. $\min\limits_G \max\limits_D V(G, D)$ picks a generator that minimize the difference.

1.4 Solve $\arg\max\limits_D V(G, D)$

Given a G, thus distribution p_G is determined. We have

$$\begin{aligned} V &= E_{x\sim p_{data}}[log(D(x))] + E_{z\sim p_z}[log(1 - D(G(z)))] \\ &= E_{x\sim p_{data}}[log(D(x))] + E_{x\sim p_G}[log(1 - D(x))] \\ &= \int_x p_{data}(x) log(D(x))dx + \int_x p_G(x) log(1 - D(x))dx \\ &= \int_x [p_{data}(x) log(D(x)) + p_G(x) log(1 - D(x))]dx \end{aligned} \tag{6}$$

Because p_{data} and p_G are both fixed, for each input x, we can easily compute the result that maximizes $V(G, D)$:

$$D^*(x) = \frac{p_{data}(x)}{p_{data}(x) + p_G(x)} \tag{7}$$

1.5 Why $\max\limits_D V(G, D)$ evaluates the difference of distributions?

$$\begin{aligned} &\max_D V(G, D) \\ &= V(G, D^*) \\ &= E_{x\sim p_{data}}[log\frac{p_{data}(x)}{p_{data}(x) + p_G(x)}] + E_{x\sim p_G}[log\frac{p_G(x)}{p_{data}(x) + p_G(x)}] \\ &= \int_x p_{data}(x) log\frac{p_{data}(x) \times \frac{1}{2}}{(p_{data}(x) + p_G(x)) \times \frac{1}{2}}dx \\ &\quad + \int_x p_G(x) log\frac{p_G(x)}{\frac{1}{2}(p_{data}(x) + p_G(x)) \times \frac{1}{2}}dx \\ &= -2log2 + \int_x p_{data}(x) log\frac{p_{data}(x)}{\frac{1}{2}(p_{data}(x) + p_G(x))}dx \\ &\quad + \int_x p_G(x) log\frac{p_G(x)}{\frac{1}{2}(p_{data}(x) + p_G(x))}dx \\ &= -2log2 + KL(p_{data}||\frac{p_{data} + p_G}{2} + KL(p_G||\frac{p_{data} + p_G}{2}) \\ &= -2log2 + JS(p_{data}||p_G) \end{aligned} \tag{8}$$

Therefore, we can see that $V(G, D^*)$ is the sum of a JS divergence and a constant $-2log2$. So it can evaluate the difference of distributions. And the smaller the $V(G, D^*)$ is, the more similar p_{data} and p_G are.

1.6 Final target - the generator

Our final aim is to achieve a generator G that have the same distribution p_G as p_{data}, which means the smallest divergence. So our optimal objective is

$$G^* = \arg\min_G \max_D V(G, D) \tag{9}$$

1.7 Tips for implementing a GAN

1. Due to the difficulty of computing the real expectation of distributions, we use the arithmetic mean of samples $\{x^(1), ..., x^{(m)}\}$ and $\{z^{(1)}, ..., z^{(m)}\}$ to approximate it:

$$\frac{1}{m}\sum_{i=1}^{m}[log(D(x^{(i)})) + log(1 - D(G(z^{(i)})))] \tag{10}$$

2. Update the generator by descending the following stochastic gradient can be inefficient:

$$\nabla_{\theta_G}\frac{1}{m}\sum_{i=1}^{m} log(1 - D(G(z^{(i)}))) \tag{11}$$

Because the equation indicates that the more real the generated image z is, the smaller the gradient is. That is, the gradient will be almost 0 at the begin of training and increase significantly when the generator is able to generate a good enough image! This will make the training inefficient and uneasy to converge. So we usually use the gradient below as an alternative. More real the image, less the gradient.

$$\nabla_{\theta_G}\frac{1}{m}\sum_{i=1}^{m} log(D(G(z^{(i)}))) \tag{12}$$

2 (In Detail) DCGAN: Deep Convolutional GAN

2.1 Architecture guidelines for stable DCGANs

1. Do not use any pooling layers. Use strided convolutional layers in discriminator network and fractional-strided convolutional layers in generator network instead.
2. Use batch normalization in both generator and discriminator network. Note that directly applying batch normalization to all layers may result in sample oscillation and instability. In order to avoid this situation, do not apply batch normalization to the output layer of generator and the input layer of discriminator.
3. Do not use any fully connected layers. We can use global pooling instead which contributes to model stability but slow down the convergence.
4. Use ReLU activation in generator network for all layers except for the output layer which uses tanh activation to project values to $[-1, 1]$.
5. Use Leaky ReLU in all layers of discriminator network.

2.2 Training details for DCGANs

1. No pre-processing. Only use tanh activation to scale the range of value into $[-1, 1]$
2. SGD
3. Batch size of 128
4. All weights are initialized from a zero-center normal distribution with standard deviation 0.02
5. The slope of Leaky ReLU is 0.2
6. Adam optimizer
7. Learning rate of 0.0002
8. Momentum term β_1 is 0.5 instead of 0.9

2.3 Validate the model capacity

In this part, the author mainly measure the capacity of discriminator network as feature extractors compared with other unsupervised learning methods.

2.3.1 CIFAR dataset

We use all convolutional layers of discriminator which comes from a pre-trained DCGAN model on Imagenet-1k. We maxpool each layer representation to produce a 4×4 spatial grid. We then flatten and concatenate these grids to form a 28672 dimensional vector. By putting the output of the feature vector into a L2-SVM linear model, we can get a classification scores for supervised learning tasks.

As a result, DCGAN+L2-SVM performs better than k-means baseline method but is not as good as the Exemplar CNN(another method to apply unsupervised learning tasks with CNN).

2.3.2 SVHN dataset

After the similar approach above, we can transfer the discriminator' convolutional layers into feature vector and put it with L2-SVM. This time, DCGAN+L2-SVM outperform the previous works when labeled data is scarce.

2.4 Visualization

In this part, the author wants to mainly validate that the DCGAN model is actually learning features and representation from the image instead of simply memorizing and fitting the input image. The demonstration is various and imaginative.

2.4.1 Walking in the latent space

Interpolation applied to latent vector z results in smooth transition on the generated images. E.g. images transfer from having windows to not having windows. This indicates the feature representation in latent vector z.

2.4.2 Visualizing discriminator features

Using the *guided backpropagation*, we can visualize the last convolutional layer of discriminator network. And we can see many base outline of specific objects like a mirror or a window. This indicates the features learned in discriminator network.

2.4.3 Visualizing generator features

The author do the following two experiments, which strongly demonstrate that there is feature hierarchy architecture in generator network.

1. **Eliminate objects by modifying the generator** - Use logistic regression to predict whether a feature activation is on a window or not based on 150 manually labeled generated images. Drop all activations that are greater than zero(indicating a window). Then, applying the same vector z, generator network will produce image without window accordingly! But the generated image is also blurrier.
2. **Vector arithmetic** - Applying vector arithmetic on input z will result in feature object combination or elimination.

3 (In Detail) WGAN: Wasserstein GAN

3.1 Problems of the original loss function in GAN models

Before WGAN, we usually use one of the two formulas below as the loss function of the generator in GAN models:

$$Loss_g = E_{x\sim P_g}[log(1 - D(x))] \tag{13}$$

$$Loss_g = E_{x\sim P_g}[-log(D(x))] \tag{14}$$

However, both of the formulas above have some problems thus leading to the instability of training GAN models.

3.1.1 Problems of $E_{x\sim P_g}[log(1 - D(x))]$

As it is proved in paper note of *GAN: Generative Adverarial Network*, when the generator is fixed, the optimal discriminator can be written as:

$$D^*(x) = \frac{P_{data}(x)}{P_{data}(x) + P_g(x)} \tag{15}$$

It is obvious that minimizing loss function $E_{x\sim P_g}[log(1 - D(x))]$ is equivalent to minimizing the formula below:

$$Loss_g = E_{x\sim P_{data}}[log(D(x))] + E_{x\sim P_g}[log(1 - D(x))] \tag{16}$$

After putting the optimal discriminator formula into the equation above, we can finally get an equivalent formula of loss function below:

$$Loss_g = 2JS(P_{data}||P_g) - 2log2 \tag{17}$$

As we can see, when using $E_{x\sim P_g}[log(1 - D(x))]$ as the loss function of the generator, we actually use the JS divergence to measure the difference of two distributions. But the JS divergence has a fatal drawback: it only measures the overlaps of two distributions. In other words, if two distributions has little overlaps, the JS divergence will be a constant close to $log2$ and provide a gradient close to 0, which makes the generator hard to improve.

On the other hand, there is actually nearly no overlaps between two distributions P_{data} and P_g due to the following two reasons:

1. **Nature of data.** The supports of distributions P_{data} and P_g are only the manifolds in the high-dimension space, because they are generated from a latent vector of 100 dimension in spite of the image size $[64, 64]$. Intuitively, two curves can hardly overlap in a 3D space. So P_{data} and P_g will have closely no overlaps in the high-dimension space.
2. **Sampling for approximation.** When we implement a GAN model, we actually do sampling to approximate the expected values. So even if two distributions P_{data} and P_g luckily have some overlaps, we, in fact, only use some sampling points the measure the overlaps. This operation leads to the debasement of computational overlaps. Also, an optimal discriminator will be strong enough to overfit and divide those points perfectly, which means that the computational JS divergence will be $log2$.

From the inference above, we can see that if a discriminator network is trained too well (thus close to the optimal one), the gradient of generator will be close to zero and make the generator stop to improve.

Therefore, if using $E_{x\sim P_g}[log(1-D(x))]$ as the loss function of the generator, we will find us in an awkward situation. On the one hand, we can not train the discriminator too little because the discriminator will be too weak to provide correct gradients for the generator to update. On the other hand, we can not train the discriminator too much because a closely optimal discriminator will lead to zero-gradient problem for the generator.

This is why training GAN models with $Loss_g = E_{x\sim P_g}[log(1-D(x))]$ is usually unstable,

3.1.2 Problems of $E_{x\sim P_g}[-log(D(x))]$

This is an optimized version put forward by *Ian Goodfellow*, but it is also not satisfying in practice. First, let us rewrite the formula of KL divergence:

$$\begin{aligned} KL(P_g||P_{data}) &= E_{x\sim P_g}[log\frac{P_g(x)}{P_{data}(x)}] \\ &= E_{x\sim P_g}[log\frac{P_g(x)/(P_g(x)+P_{data}(x))}{P_{data}(x)/(P_g(x)+P_{data}(x))}] \\ &= E_{x\sim P_g}[log\frac{1-D^*(x)}{D*(x)}] \\ &= E_{x\sim P_g}[log(1-D^*(x))] - E_{x\sim P_g}[log(D^*(x))] \end{aligned} \tag{18}$$

Therefore, if we got an optimal discriminator $D^*(x)$, the loss function of the generator can be rewritten as below:

$$\begin{aligned} Loss_g &= E_{x\sim P_g}[-log(D(x))] \\ &= KL(P_g||P_{data}) - E_{x\sim P_g}[log(1-D^*(x))] \\ &= KL(P_g||P_{data}) - 2JS(P_{data}||P_g) + 2log2 + E_{x\sim P_{data}}[log(D^*(x))] \end{aligned} \tag{19}$$

Because the last two terms have no correlations to P_g, so the equivalent formula can be:

$$Loss_g = KL(P_g||P_{data}) - 2JS(P_{data}||P_g) + 2log2 \tag{20}$$

As we can see, the generator tries to minimize the KL divergence and maximizing the JS divergence at the same time. That is, the generator tries to pull two distributions closer and push two distributions farther at the same time, which sounds ridiculous. Also, the KL divergence, similar to the JS divergence, is noncontinuous, which can leads to zero-gradient situation.

More severely, the KL divergence is asymmetric. And let us consider the following two situations:

- When $P_g(x) \to 0$ and $P_{data}(x) \to 1$, $KL(P_G||P_{data}) \to 0$
- When $P_g(x) \to 1$ and $P_{data}(x) \to 0$, $KL(P_G||P_{data}) \to \infty$

The former situation is that the generator **can not generate a real image**. The latter one is that the generator **generates a nonexistent image**. But the model will punish the latter one severely but do nothing to the former one. Therefore, the generate will tend to generate repeated but more 'safe' images instead of various images, which is the so-called **mode collapse**.

3.1.3 Section summary

The primary reason about the training instability of GAN models comes from two aspects. One is that our divergence measurements are inappropriate. Another is that two distributions P_g and P_{data} can hardly have overlaps in the high-dimensional space because they are manifolds. Thus, WGAN is proposed to solve these fundamental problems.

3.2 Wasserstein distance

Wasserstein distance is also called Earth-Mover distance. Here is the definition of it:

$$W(P_{data}, P_g) = \inf_{\gamma \sim \Pi(P_{data}, P_g)} E_{(x,y)\sim\gamma}[||x - y||] \tag{21}$$

Intuitively, we can see γ as an earth-move plan and $E_{(x,y)\sim\gamma}[||x - y||]$ is the average cost for move 'earth' from P_{data} to P_g. And Wasserstein distance seeks for the minimal cost out of all possible moving plans.

Compared to the KL or JS divergence, Waseserstein distance is smooth and continuous, and it can measure the distance between two distributions even there are no overlaps.However, the inf is impossible to solve. By using some theorems, the author transforms the formula above into the equation below (proving process omitted due to the complexity):

$$W(P_{data}, P_g) = \frac{1}{K} \sup_{||f||_L \leq K} E_{x\sim P_{data}}[f(x)] - E_{x\sim P_g}[f(x)] \tag{22}$$

The character L represents the Lipschitz Continuity, which means that there exists a real constant $K \geq 0$ such that, for all x_1 and x_2 in the domain:

$$|f(x_1) - f(x_2)| \leq K|x_1 - x_2| \tag{23}$$

The Lipschitz Continuity constrains a continuous function to be smooth enough and should not oscillate widely. We can use a parameter w to define a series of Lipschitz functions f_w. The formula are written as:

$$K \times W(P_{data}, P_g) = \max_{w:|f_w|_L \leq K} E_{x\sim P_{data}}[f(x)] - E_{x\sim P_g}[f(x)] \tag{24}$$

Then we can use a neuron network to represent the function f_w. They are both smooth and the neuron network have the capacity to approximate all possible continuous functions, which indicate that the formula above is solvable by using our deep learning techniques. It does not matter how much the constant K is because it will always provide the correct gradient direction if K is a finite number.

3.3 Weight clipping

In order to constrain the network f_w to satisfy the Lipschitz continuity, the author proposes a method called **weight clipping**, which forces every single parameter in the network to be in range of $[-c, c]$. Therefore, the network will meets the condition although it might miss some function series.

3.4 Architecture of WGAN

Now, we can construct a discriminator network f_w to maximize the formula:

$$E_{x\sim P_{data}}[f(x)] - E_{x\sim P_g}[f(x)] \tag{25}$$

Therefore, we can define the loss function that the discriminator network tries to minimize (just the inverse version):

$$Loss_d = E_{x\sim P_g}[f(x)] - E_{x\sim P_{data}}[f(x)] \tag{26}$$

When trying to minimize the formula above, the discriminator will gradually approximate the ground true Waseserstein distance. The smaller the Waseserstein distance is, the better the generator performs. Then, based of this approximate Waseserstein distance, the generator will try to minimize the Waseserstein distance. So the loss function is defined as (the 'P_{data}' term has no relation to the geenerator so it is erased):

$$Loss_g = -E_{x \sim P_g}[f(x)] \tag{27}$$

Additionally, we can see that the discriminator in WGAN are doing regression tasks (approximating Waseserstein distance) instead of binary classification tasks in the traditional GAN, so the last **sigmoid** layer (proposed in DCGAN) is removed.

Also, the author empirically suggests that we would better use RMSProp or SGD instead of momentum-based optimization algorithm like Adam. This tip comes from the extensive experiments taken by the author, so I am not sure about the correctness. Maybe it worth a try in the implementation.

3.5 Total summary of WGAN

1. Remove the **sigmoid** layer in the discriminator
2. Remove the **log** in the loss function of discriminator and generator
3. Apply weight clipping
4. Try non-momentum-based optimization algorithm like RMSProp or SGD.

4 (In Detail) WGAN-GP: Wasserstein GAN with gradient penalty

4.1 What is wrong with WGAN ?

When implementing the WGAN, we can still find it hard to train and the training is quite slow. The problem stems from crucial part of WGAN - how to apply Lipschitz Continuity. The Lipschitz Continuity is defined as:

$$|D(x_1) - D(x_2)| \leq K|x_1 - x_2| \tag{28}$$

or equivalently:

$$||\nabla_x D(x)|| \leq K \tag{29}$$

In WGAN, we apply weight clipping to indirectly achieve the Lipschitz Condition, which forces every parameter in the network to be in range of $[-c, c]$. So that the gradient of the network will also be limited, thus approximate the Lipschitz Condition. But weight clipping will meet two severe problems.

First, as we can see from the loss of the discriminator $Loss_d = E_{x \sim P_g}[D(x)] - E_{x \sim P_{data}}[D(x)]$, the discriminator tends to make the score of real images higher and the score of generated images lower. Therefore, in weight clipping, the discriminator tends to set every single parameter to be either $-c$ or c. For example, we can see the value distribution of parameters in Figure 1 if set c to be 0.01.

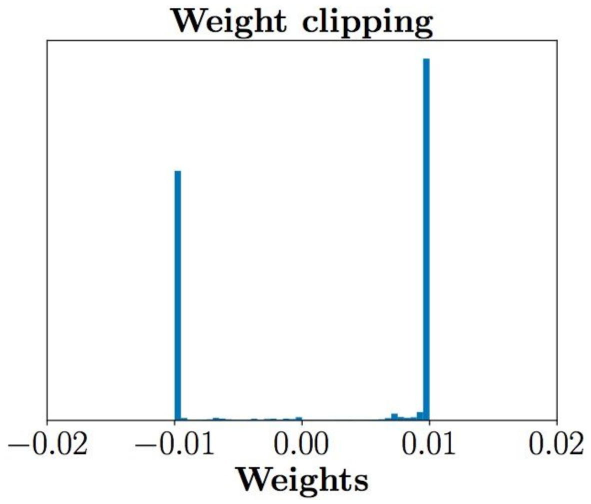

Figure 1: Weight distribution of weight clipping

This polarization will make the discriminator too weak thus leading to a bad performance. Additionally, weight clipping will lead to gradient vanishing or gradient exploding problems. Thus, the author provide an optimized method to replace weight clipping.

4.2 Gradient penalty

In a word, Lipschitz Continuity constrains the gradient of a function to be less than a constant K. So why not add this restriction into the loss function? So we modify the loss of the discriminator into the version below:

$$Loss_d = E_{x \sim P_g}[D(x)] - E_{x \sim P_{data}}[D(x)] + \lambda E_{x \sim \omega}[||\nabla_x D(x)|| - K]^2 \tag{30}$$

where ω is the sample space and K is the Lipschitz Constant which is set to be 1 in practice. We will do sampling to approximate the three expected values in the formula in practice, but it is unreasonable to sample from the entire sample space ω. As the author suggests, we can only focus on P_g, P_{data} and the area between them. So, we define a new distribution $P_{\hat{x}}$:

$$\hat{x} = \epsilon x_{data} + (1-\epsilon)x_g, \quad \epsilon \sim Uniform(0,1) \tag{31}$$

Therefore, we can write down the loss of the discriminator in WGAN-GP:

$$Loss_d = E_{x\sim P_g}[D(x)] - E_{x\sim P_{data}}[D(x)] + \lambda E_{x\sim P_{\hat{x}}}[||\nabla_x D(x)|| - 1]^2 \tag{32}$$

As a reminder, we are applying sample-independent gradient penalty, so batch normalization should not be used because it will bring in correlations between samples. So the author chooses layer normalization instead.

Because the formula above needs to compute the second gradient of the loss, some blogs propose that we can replace it with a finite difference.

$$Loss_d = E_{x\sim P_g}[D(x)] - E_{x\sim P_{data}}[D(x)] + \lambda E_{x_1,x_2\sim P_{\hat{x}}}[\frac{D(x_1)-D(x_2)}{x_1-x_2} - 1]^2 \tag{33}$$

Finally, for comparison, Figure 2 shows the weight distribution of gradient penalty.

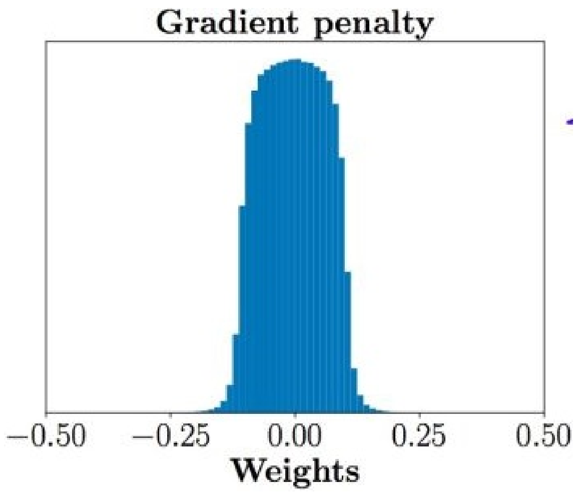

Figure 2: Weight distribution of gradient penalty

5 (In Detail) StyleGAN

5.1 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 3.

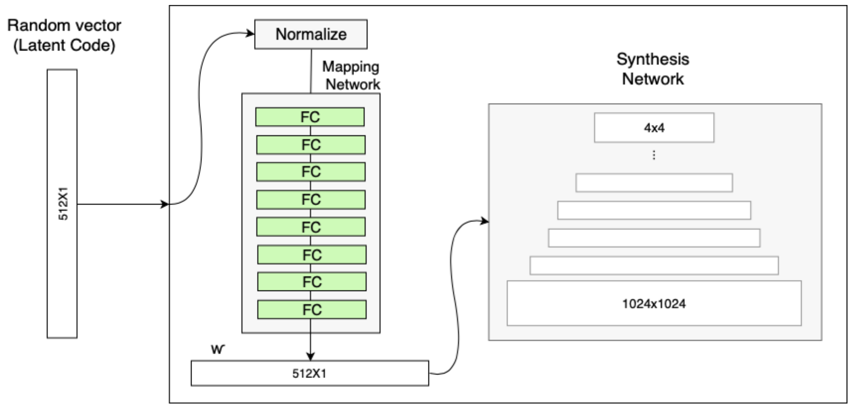

Figure 3: Mapping Network

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.

5.2 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.

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

$$y = (y_s, y_b) = f(w)haiyou \tag{34}$$

$$AdaIN(x_i, y) = y_s \frac{x_i - \mu(x)}{\sigma(x)} + y_b \tag{35}$$

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

5.3 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 4.

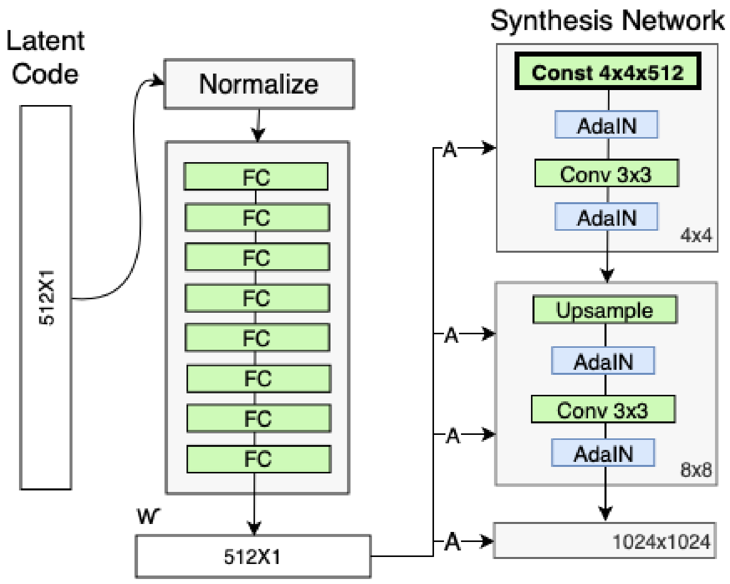

Figure 4: Basic Architecture of StyleGAN

5.4 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).

5.5 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 5.

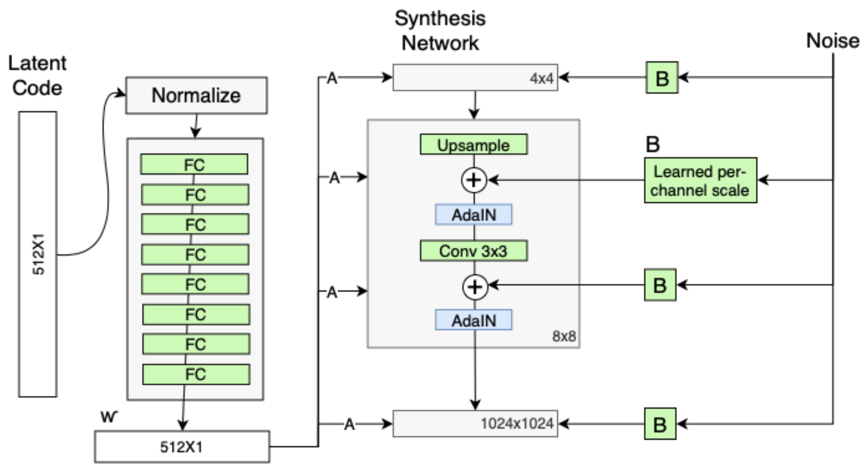

Figure 5: Final Architecture of StyleGAN

6 (In Detail) BiGAN: Bidirection GAN

Before all, note that there is a model called ALI, short for *Adversarially Learned Inference*, it has exactly the same idea as BiGAN and both of these two papers have been recieved by ICLR 2017. But we prefer the name BiGAN. Maybe because it contains the word 'GAN'.

6.1 Architecture of BiGAN

In a GAN model, there are 2 spaces:

- **the feature space** - where latent vectors z are sampled from
- **the image space** - P_{data} and P_g are both sub-distributions of the image space.

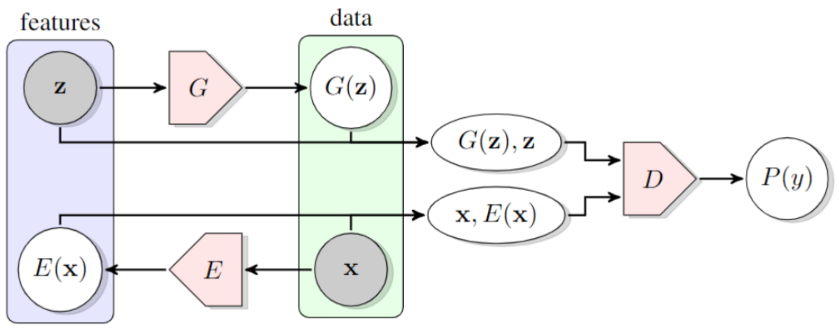

Figure 6: Architecture of BiGAN

As we can see in Figure 6, In BiGAN, we will train a **independent** generator: an encoder that transform inputs from the *image space* to the *feature space* (latent space), and a decoder that transform inputs from the *feature space* to the *image space*. Here the bolded word **independent** means that two network will not communicate in the generator part. That is, during the training, we will not **reconstruct** an image or a latent vector to apply some similar loss like VAE.

Therefore, the encoder and decoder will only communicate in the discriminator part. The discriminator network takes inputs combined of samples from both the *image space* and the *feature space*. This combined input has two versions: a real sampled feature z and a generated image $G(z)$, or a real sampled image x and an encoded feature $E(x)$. And the discriminator aims to identify which type the combined input belongs to.

6.2 How do G and E communicate ?

As we known, combined inputs for the discriminator have two different types: $(G(z), z)$, or $(x, E(x))$. We can see these two types as two different distributions. The former one is the joint distribution of $P_{G(z)}$ and P_z. The latter one is the joint distribution of P_x and $P_{E(x)}$.

Therefore, the task of the discriminator is to decide which joint distribution the input belongs to. So the encoder $E(x)$ and the decoder $G(z)$ have to work together to fool the discriminator. The discriminator can easily make the correct answer if one of them has a bad performance.

If BiGAN finally achieves the optimal solution, two joint distributions will be exactly the same. Then the encoder and the decoder can be put together as an autoencoder to generate either reconstructed images $G(E(x))$ or reconstructed features $E(G(z))$.

The theoretical proof for these intuitions are so complex that I omit it here. Anyone interested in the proof can refer to the appendix of the original paper Adversarial Feature Learning.

The BiGAN training objective is defined as a min-imax objective

$$\min_{G,E} \max_{D} V(D, E, G) \tag{36}$$

where

$$V(D,E,G) = E_{x\sim P_{data}}[E_{z\sim P_E}[log(D(x,z))]] + E_{z\sim P_z}[E_{x\sim P_G}[log(1-D(x,z)]] \qquad (37)$$

6.3 What is the difference of BiGAN and VAEGAN ?

The idea of BiGAN and VAEGAN both comes from Autoencoder and GAN. The only difference is that in VAEGAN, the encoder and decoder will communicate in the generator part by reconstruction, but in BiGAN they will communicate in the discriminator part.

The encoder and decoder will be exactly the same if both models reach the global optimal solution. But obviously we can never achieve that in deep learning. So the actual outputs of two models will differ when we try to reconstruct an image. If we input an image of a bird. The reconstructed image from VAEGAN will still be that bird but kind of blurry. The reconstructed image from BiGAN will be sharper but maybe another bird that differ from the input one. This is the difference between VAEGAN and BiGAN.

7 (In Detail) BigBiGAN: Big Bidirection GAN

As the name suggests, BigBiGAN is a combination of BigGAN and BiGAN. Because the original BiGAN model is based on the generator proposed by DCGAN, which have relatively bad performance in image generation, thus leading to a bad performance in feature extracting. So BigBiGAN replaces the generator with BigGAN and, together with some modifications, achieves the state of are in unsupervised feature learning.

Apart from replacing DCGAN with BigGAN, there is only one different architecture detail compared to BiGAN and BigGAN. As we can see in Figure 7, the discriminator have three parts F, H and J, which respectively represent three different networks.

The F network is just like the original discriminator in traditional GAN models. It takes an image as an input and decides it is fake or not. The H network, in the other hand, will distinguishes encoded latent vectors from the sampled ones. And the J is similar to the discriminator in BiGAN. It will take a joint input and identify its origin - $(x, E(x))$ or $(G(z), x)$.

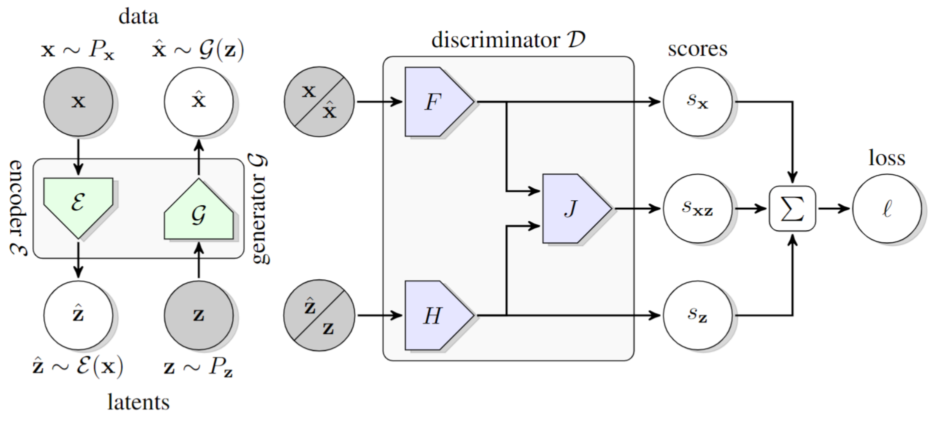

Figure 7: Architecture of BigBiGAN

All three network above will give independent scores - s_x, s_z, s_{xz}. The score for each sample can be computed by summing three scores up. Then we can define the loss of the generator and discriminator based on the per-sample scores:

$$l_{E\&G}(x, z, y) = y(s_x + s_z + s_{xz}), \quad y \in \{-1, +1\} \tag{38}$$

$$Loss_{E\&G} = E_{x \sim P_{data}, \tilde{z} \sim P_{E(x)}}[l_{E\&G}(x, \tilde{z}, +1)] + E_{z \sim P_z, \tilde{x} \sim P_{G(z)}}[-l_{E\&G}(\tilde{x}, z, -1)] \tag{39}$$

$$l_D(x, z, y) = h(y \times s_x) + h(y \times s_z) + h(y \times s_{xz}), \quad y \in \{-1, +1\} \tag{40}$$

$$Loss_D = E_{x \sim P_{data}, \tilde{z} \sim P_{E(x)}}[l_D(x, \tilde{z}, +1)] + E_{z \sim P_z, \tilde{x} \sim P_{G(z)}}[-l_D(\tilde{x}, z, -1)] \tag{41}$$

where $h(t) = \max(0, 1 - t)$ is a hinge function used to regularize the discriminator, also used in BigGAN. Then other architecture details are the same as it is in BiGAN or BigGAN. Together with some training techniques like *truncation tricks*, the performance of BigBiGAN is surprising. It does not only defeat BigGAN in image generation, but also outperforms those unsupervised representation learning algorithms on ImageNet.

That is all for BigBiGAN. **Bigger**, **larger**, and **stronger**.

8 (Briefly Reading) Summary of GAN models

8.1 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.

8.2 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**:

$$F(G(x)) \approx x, x \in X \tag{42}$$

$$G(F(y)) \approx y, y \in Y \tag{43}$$

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

$$L_{GAN}(G, D_Y, X, Y) = E_y[log(D_Y(y))] + E_x[log(1 - D_Y(G(x))] \tag{44}$$

$$L_{cycle} = E_x[||F(G(x)) - x||_1] + E_y[||G(F(y)) - y||_1] \tag{45}$$

$$Loss = L_{GAN} + L_{cycle} \tag{46}$$

8.3 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 8.

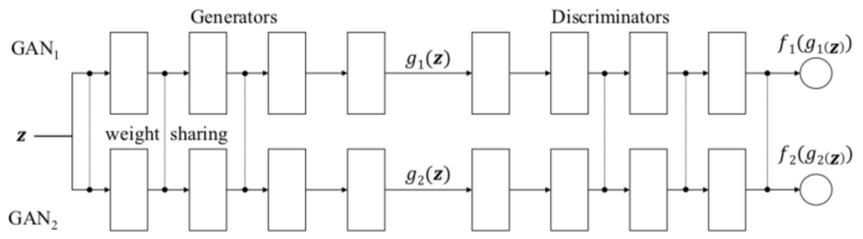

Figure 8: CoGAN

8.4 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 9.

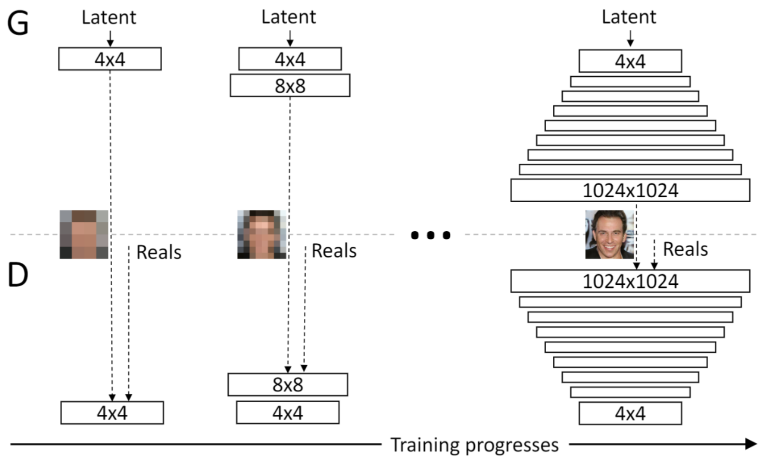

Figure 9: ProGAN

8.5 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.

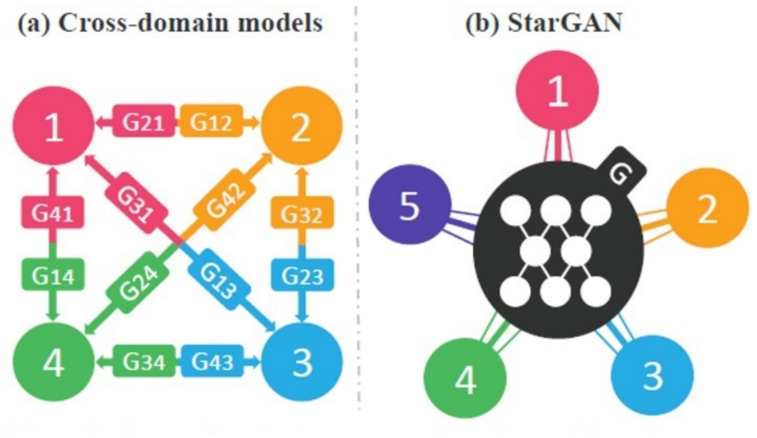

Figure 10: StarGAN

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 11.

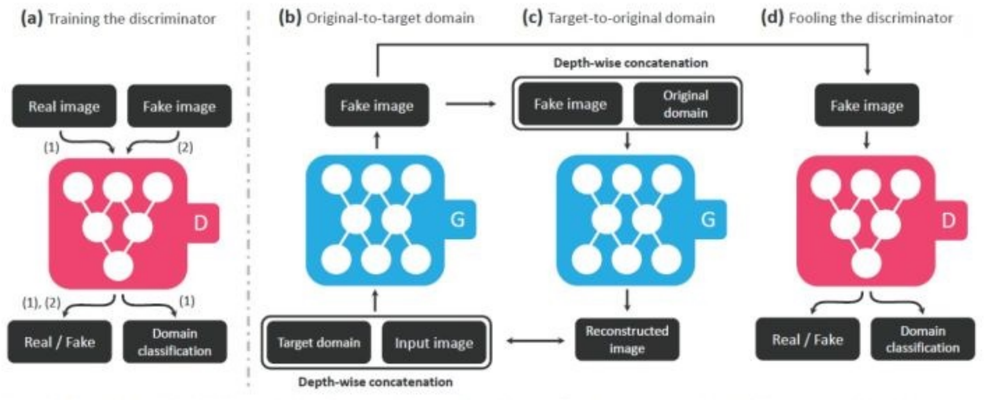

Figure 11: Architecture of StarGAN

8.6 InfoGAN: Information GAN

As shown in Figure 12, InfoGAN will first separate the latent vector into two part c and $z^{'}$. Part c is expected to contain the encoded information of the generated images x and $z^{'}$ represents the variant part which can not be learned during the training.

The generator, acting as an inverse encoder, will transform the $c + z^{'}$ vector into an image x. A additional classifier network, acting as an inverse decoder, will predict the code c that generates x. Also a discriminator network is necessary for the model to generate a realistic image instead of an image that is easy to predict c but totally unrealistic.

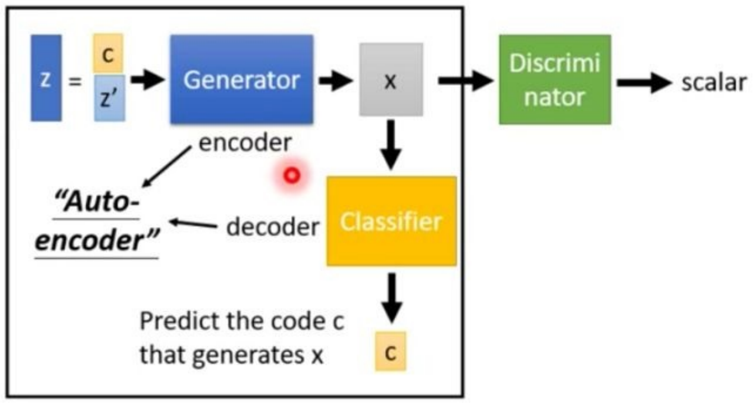

Figure 12: Architecture of InfoGAN

8.7 SAGAN: Self-Attension GAN

Since GAN models use transposed convolutions to scan feature maps, they only have access to nearby information. But when painting a picture, we should focus on the overall layout of the image instead of only looking at the local parts. SAGAN tries to handle this problem.

Attention is a useful technique that can be apply to various specific deep learning areas and GAN is one of them. As a brief summary, an attention will have three matrices **Q**uery, **K**ey and **V**alue. And the query and key decide how much the value can speak to the outer world (shown in Figure 13). When it comes to **self-attention**, it means that the query, key and value are all same.

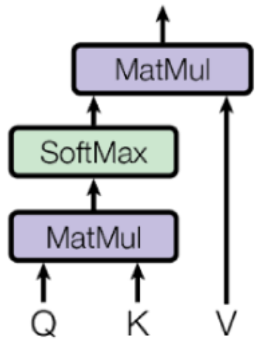

Figure 13: Attention

We will apply self-attention to every layer of both generator and discriminator network. Figure 14 shows the architecture of SAGAN. And the construction of the attention map is shown in Figure 15. By combining the self-attention part, SAGAN can take care of the overall layout of the image and generate more realistic images.

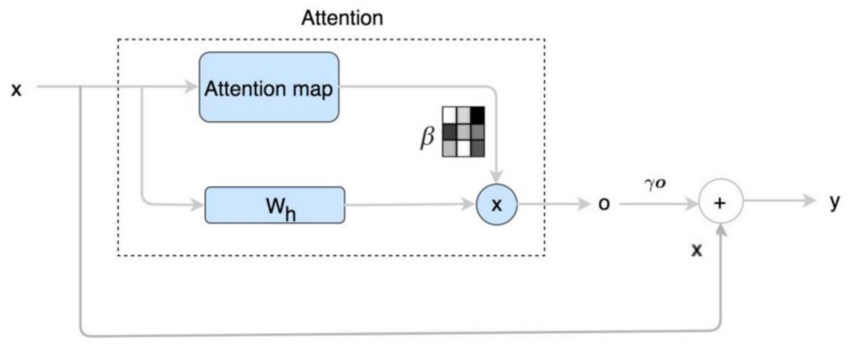

Figure 14: Architecture of SAGAN

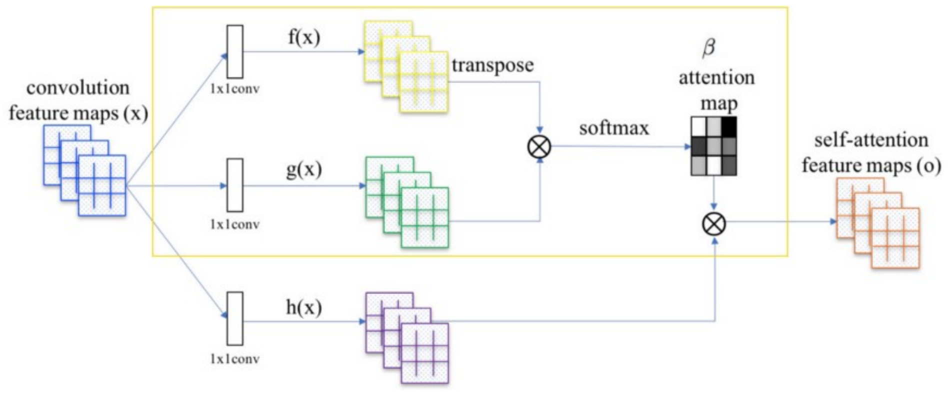

Figure 15: Attention map of SAGAN

8.8 VAEGAN: Variational Autoencoder GAN

As shown in Figure 16, VAEGAN combines VAE and GAN. The real images sampled from P_{data} will first be put into an encoder to produce a latent vector z. Then the generator, acting as a decoder, will decode the latent vector z to generate a reconstructed image $\tilde{x}$. Next, the discriminator aims to distinguish the reconstructed images $\tilde{x}$ from the real images x.

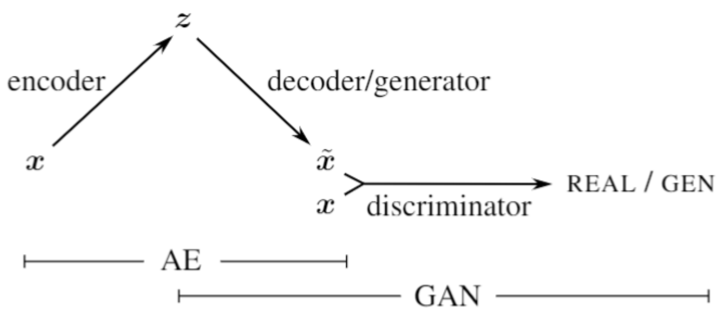

Figure 16: Architecture of VAEGAN