Chapter 13 Generative Adversarial Networks

What I cannot create, I have not yet fully understood.

− Richard Feynman

Before the invention of the Generative Adversarial Network (GAN), the variational autoencoder was considered to be theoretically complete and simple to implement. It is very stable when trained using neural networks, and the resulting images are more approximate, but the human eyes can still easily distinguish real pictures and machine-generated pictures.

In 2014, Ian Goodfellow, a student of Yoshua Bengio (the winner of the Turing Award in 2018) at Université de Montréal, proposed the GAN [1], which opened up one of the hottest research directions in deep learning. From 2014 to 2019, GAN research has been steadily advancing, and research successes have been reported frequently. The effect of the latest GAN algorithm on image generation has reached a level that is difficult to distinguish with the naked eyes, which is really exciting. Due to the invention of GAN, Ian Goodfellow was awarded the title of Father of GAN, and was granted the 35 Innovators Under 35 award by the Massachusetts Institute of Technology Review in 2017. Figure 13.1 shows that from 2014 to 2018, the GAN model achieved the effect of book generation. It can be seen that both the size of the picture and the fidelity of the picture have been greatly improved.

Figure 0.1 GAN generated image effect from 2014 to 2018 ①

Next, we will start from the example of game learning in life, step by step to introduce the design ideas and model structure of the GAN algorithm.

13.1 Examples of game learning

We use the growth trajectory of a cartoonist to vividly introduce the idea of GAN. Consider a pair of twin brothers, called G and D. G learns how to draw cartoons, and D learns how to appreciate paintings. The two brothers at young ages only learned how to use brushes and papers. G drew an unknown painting, as shown in Figure 13.2(a). At this time, D's discriminating ability is not high, so D thinks G's work is OK, but the main character is not clear enough. Under D's

① Image source: https://twitter.com/goodfellow_ian/status/1084973596236144640?lang=en

guidance and encouragement, G began to learn how to draw the outline of the subject and use simple color combinations.

A year later, G improved the basic skills of painting, and D also initially mastered the ability to identify works by analyzing masterpieces and the works of G. At this time D feels that G's work has the main character, as shown in Figure 13.2(b), but the use of color is not mature enough. A few years later, G's basic painting skills have been very solid, and he can easily draw paintings with bright subjects, appropriate color matching and high fidelity, as shown in Figure 13.2(c), but D also observes the differences between G and other masterpieces, and improved the ability to distinguish paintings. At this time, D felt that G's painting skills have matured, but his observation of life is not enough. G's work does not convey the expression and some details are not perfect. After a few more years, G's painting skills have reached the point of perfection. The details of the paintings are perfect, the styles are very different and vivid, just like a master level, as shown in Figure 13.2(d). Even at this time, D's discrimination skills are quite excellent. It is also difficult for D to distinguish G from other masterpieces.

The growth process of the above-mentioned painters is actually a common learning process in life, through the game of learning between the two sides and mutual improvement, and finally reach a balance point. The GAN network draws on the idea of game learning and sets up two sub-networks: a generator G responsible for generating samples and a discriminator D responsible for authenticating. The discriminator D learns how to distinguish between true and false by observing the difference between the real sample and the sample produced by the generator G, where the real sample is true and the sample produced by the generator G is false. The generator G is also learning. It hopes that the generated samples can be recognized by the discriminator D as true. Therefore, the generator G tries to make the samples it generates be considered as true by discriminant D. The generator G and the discriminator D play a game with each other and improve together until they reach an equilibrium point. At this time, the samples generated by the generator G are very realistic, making the discriminator D difficult to distinguish between true and false.

(a) (b) (c) (d)

Figure 0.2 Sketch of the painter's growth trajectory

In the original GAN paper, Ian Goodfellow used another vivid metaphor to introduce the GAN model: The function of the generator network G is to generate a series of very realistic counterfeit banknotes to try to deceive the discriminator D, and the discriminator D learns the difference between the real money and the counterfeit banknotes generated by generator G to master the banknote identification method. These two networks are synchronized in the process of mutual games, until the counterfeit banknotes produced by the generator G are very reall, and even the discriminator D can barely distinguish.

This idea of game learning makes the network structure and training process of GAN slightly different from the previous network model. Let's introduce the network structure and algorithm principle of GAN in detail below.

13.2 GAN principle

Now we will formally introduce the network structure and training methods of GAN.

13.2.1 Network structure

GAN contains two sub-networks: the generator network (referred to as G) and the discriminator network (referred to as D). The generator network G is responsible for learning the true distribution of samples, and the discriminator network D is responsible for distinguish the samples generated by the generator network from the real samples.

Generator $\mathrm{G}(\boldsymbol{z})$ The generator network G is similar to the function of decoder of the autoencoder. The hidden variables $\boldsymbol{z}\sim p_{\boldsymbol{z}}(\cdot)$ are sampled from the prior distribution $p_{\boldsymbol{z}}(\cdot)$. The generated sample $\boldsymbol{x}\sim p_g(\boldsymbol{x}|\boldsymbol{z})$ is obtained by the parameterized distribution $p_g(\boldsymbol{x}|\boldsymbol{z})$ of the generator network G, as shown in Figure 13.3. The prior distribution $p_{\boldsymbol{z}}(\cdot)$ of the hidden variable $\boldsymbol{z}$ can be assumed to be a known distribution, such as a multivariate uniform distribution $z\sim\mathrm{Uniform}(-1,1)$.

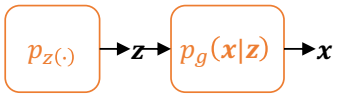


Figure 0.3 Generator G

$p_g(\boldsymbol{x}|\boldsymbol{z})$ can be parameterized by a deep neural network. As shown in Figure 13.4 below, the hidden variable $\boldsymbol{z}$ is sampled from the uniform distribution $p_{\boldsymbol{z}}(\cdot)$, and then sample $\boldsymbol{x}_f$ is obtained from the $p_g(\boldsymbol{x}|\boldsymbol{z})$ distribution. From the perspective of input and output, the function of the generator G is to convert the hidden vector $\boldsymbol{z}$ into a sample vector $\boldsymbol{x}_f$ through a neural network, and the subscript f represents fake samples.

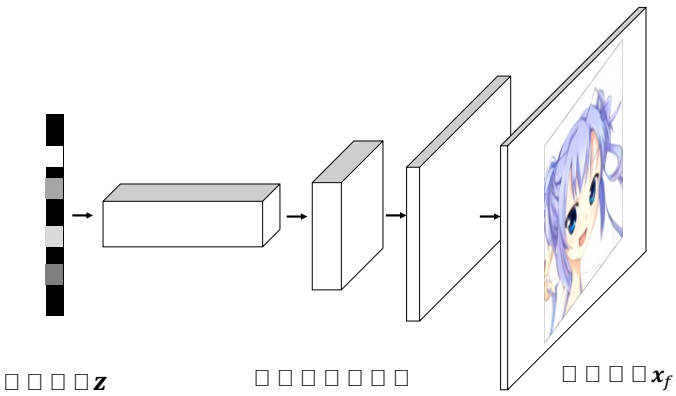


Figure 0.4 Generator network composed of transposed convolution

Discriminator $\mathrm{D}(\boldsymbol{x})$ The function of the discriminator network is similar to that of the ordinary binary classification network. It accepts a data set of input sample $\boldsymbol{x}$, including samples $\boldsymbol{x}_r\sim p_r(\cdot)$ sampled from the real data distribution $p_r(\cdot)$, and also includes fake samples sampled from the generator network $\boldsymbol{x}_f\sim p_g(\boldsymbol{x}|\boldsymbol{z})$. $\boldsymbol{x}_r$ and $\boldsymbol{x}_f$ together form the training data set of the discriminator network. The output of the discriminator network is the probability of $\boldsymbol{x}$ belonging

to the real sample $P(\boldsymbol{x}$ is real $|\boldsymbol{x})$. We label all the real samples $\boldsymbol{x}_r$ as true (1), and all the samples $\boldsymbol{x}_f$ generated by the generator network are labeled as false (0). The error between the predicted value of the discriminator network D and the label is used to optimize the discriminator network parameters as shown in Figure 13.5.

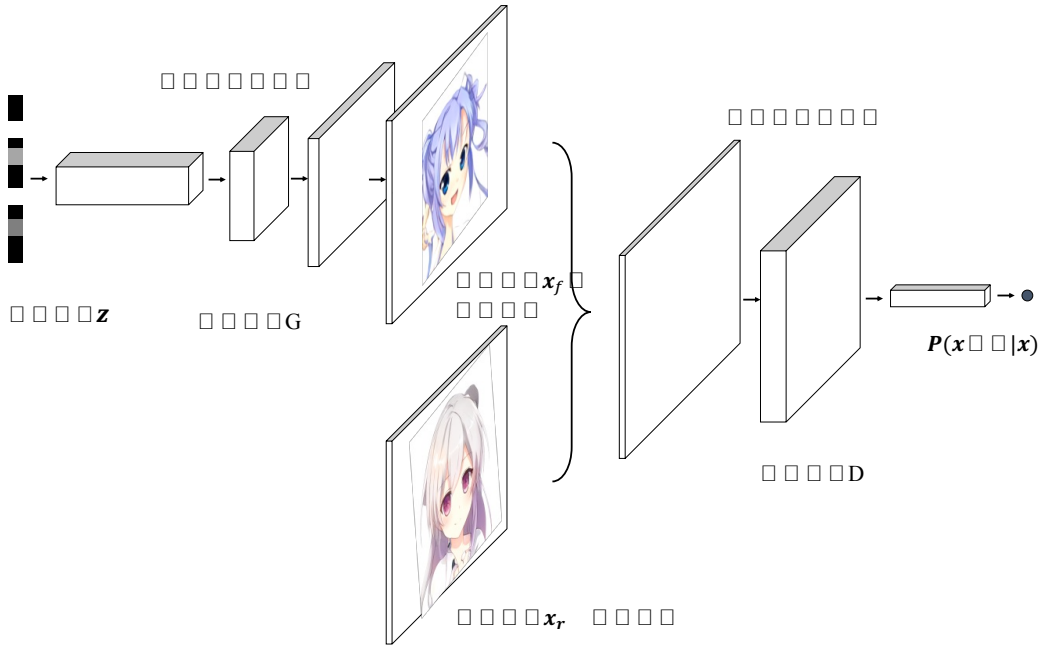


Figure 0.5 Generator network and discriminator network

13.2.2 Network training

The idea of GAN game learning is reflected in its training method. Since the optimization goals of generator G and discriminator D are different, they cannot be the same as the previous network model training, and only one loss function is used. Let us introduce how to train the generator G and the discriminator D respectively.

For the discriminator network D, its goal is to be able to distinguish the real sample $\boldsymbol{x}_r$ from the fake sample $\boldsymbol{x}_f$. Taking picture generation as an example, its goal is to minimize the cross-entropy loss function between the predicted value and the true value of the picture:

$$\min_{\theta} \mathcal{L} = \mathrm{CE}\big(D_\theta(\boldsymbol{x}_r), y_r, D_\theta(\boldsymbol{x}_f), y_f\big)$$

where $D_\theta(\boldsymbol{x}_r)$ represents the output of the real sample $\boldsymbol{x}_r$ in the discriminant network D_θ, θ is the parameter set of the discriminator network, $D_\theta(\boldsymbol{x}_f)$ is the output of the generated sample $\boldsymbol{x}_f$ in the discriminator network, and y is the label of $\boldsymbol{x}_r$. Because the real sample is labeled as true, So $y_r = 1$. y_f is the label of $\boldsymbol{x}_f$ of the generated sample. Since the generated sample is labeled as false, $y_f = 0$. The CE function represents the cross-entropy loss function CrossEntropy. The cross entropy loss function of the two classification problem is defined as:

$$\mathcal{L} = -\sum_{\boldsymbol{x}_r \sim p_r(\cdot)} \log D_\theta(\boldsymbol{x}_r) - \sum_{\boldsymbol{x}_f \sim p_g(\cdot)} \log\left(1 - D_\theta(\boldsymbol{x}_f)\right)$$

Therefore, the optimization goal of the discriminator network D is:

$$\theta^* = \arg\min_{\theta} -\sum_{\boldsymbol{x}_r \sim p_r(\cdot)} \log D_\theta(\boldsymbol{x}_r) - \sum_{\boldsymbol{x}_f \sim p_g(\cdot)} \log\left(1 - D_\theta(\boldsymbol{x}_f)\right)$$

Convert $\min_{\theta} \mathcal{L}$ to $\max_{\theta} -\mathcal{L}$, and write it in the expectation form:

$$\theta^* = \arg\max_{\theta} \mathbb{E}_{\boldsymbol{x}_r \sim p_r(\cdot)} \log D_\theta(\boldsymbol{x}_r) + \mathbb{E}_{\boldsymbol{x}_f \sim p_g(\cdot)} \log\left(1 - D_\theta(\boldsymbol{x}_f)\right)$$

For the generator network $\mathrm{G}(\boldsymbol{z})$, we hope that $\boldsymbol{x}_f = G(\boldsymbol{z})$ can deceive the discriminator network D well, and the output of the fake sample $\boldsymbol{x}_f$ is as close to the real label as possible. That is to say, when training the generator network, it is hoped that the output $D\big(G(\boldsymbol{z})\big)$ of the discriminator network is as close to 1 as possible, and the cross-entropy loss function between $D\big(G(\boldsymbol{z})\big)$ and 1 is minimized:

$$\min_{\phi} \mathcal{L} = \mathrm{CE}\left(D\left(G_\phi(\boldsymbol{z})\right), 1\right) = -\log D\left(G_\phi(\boldsymbol{z})\right)$$

Convert $\min_{\phi} \mathcal{L}$ to $\max_{\phi} -\mathcal{L}$, and write it in the expectation form:

$$\phi^* = \arg\max_{\phi} \mathbb{E}_{\boldsymbol{z} \sim p_z(\cdot)} \log D\left(G_\phi(\boldsymbol{z})\right)$$

It can be equivalently transformed into:

$$\phi^* = \arg\min_{\phi} \mathcal{L} = \mathbb{E}_{\boldsymbol{z} \sim p_z(\cdot)} \log[1 - D(G_\phi(\boldsymbol{z}))]$$

Where ϕ is the parameter set of the generator network G, and the gradient descent algorithm can be used to optimize the parameters ϕ.

13.2.3 Unified objective function

We can merge the objective functions of the generator and discriminator networks and write it in the form of a min-max game:

$$\min_{\phi} \max_{\theta} \mathcal{L}(D, G) = \mathbb{E}_{\boldsymbol{x}_r \sim p_r(\cdot)} \log D_\theta(\boldsymbol{x}_r) + \mathbb{E}_{\boldsymbol{x}_f \sim p_g(\cdot)} \log\left(1 - D_\theta(\boldsymbol{x}_f)\right)$$

$$= \mathbb{E}_{\boldsymbol{x} \sim p_r(\cdot)} \log D_\theta(\boldsymbol{x}) + \mathbb{E}_{\boldsymbol{z} \sim p_z(\cdot)} \log\left(1 - D_\theta\left(G_\phi(\boldsymbol{z})\right)\right) \tag{0-1}$$

The algorithm is as follows:

Algorithm 1：GAN training algorithm

Randomly initialize parameters $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$

repeat

  for k times **do**

    Randomly sample hidden vectors $\boldsymbol{z}\sim\boldsymbol{p_z}(\cdot)$

    Randomly sample of real samples $\boldsymbol{x_r}\sim\boldsymbol{p_r}(\cdot)$

    Update the D network according to the gradient descent algorithm:

$$\nabla_{\boldsymbol{\theta}}\mathbb{E}_{\boldsymbol{x_r}\sim\boldsymbol{p_r}(\cdot)}\log\boldsymbol{D_\theta}(\boldsymbol{x_r})+\mathbb{E}_{\boldsymbol{x_f}\sim\boldsymbol{p_g}(\cdot)}\log\left(1-\boldsymbol{D_\theta}(\boldsymbol{x_f})\right)$$

  Randomly sample hidden vectors $\boldsymbol{z}\sim\boldsymbol{p_z}(\cdot)$

  Update the G network according to the gradient descent algorithm:

$$\nabla_{\boldsymbol{\phi}}\mathbb{E}_{\boldsymbol{z}\sim\boldsymbol{p_z}(\cdot)}\log\left(1-\boldsymbol{D_\theta}(\boldsymbol{G_\phi}(\boldsymbol{z}))\right)$$

  end for

until the number of training rounds meets the requirements

output: Trained generator $\boldsymbol{G_\phi}$

13.3 Hands-on DCGAN

In this section, we will complete the actual generation of cartoon avatar images. Refer to the network structure of DCGAN [2], where the discriminator D is implemented by a common convolutional layer, and the generator G is implemented by a transposed convolutional layer, as shown in Figure 13.6 .

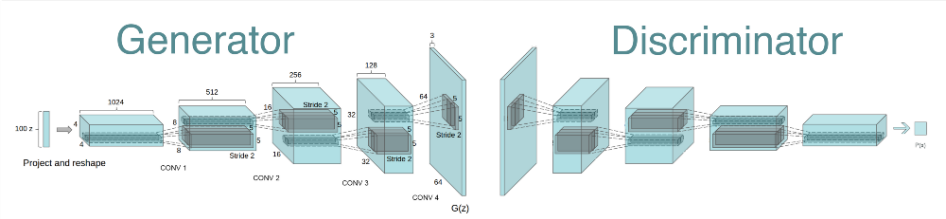


Figure 0.6 DCGAN Network structure

13.3.1 Cartoon Avatar Dataset

Here we use a data set of cartoon avatars, a total of 51,223 pictures, without annotation information. The main body of the pictures have been cropped, aligned and uniformly scaled to a size of 96×96. Some samples are shown in Figure 13.7.

Figure 0.7 Cartoon Avatar Dataset

For customized datasets, you need to complete the data loading and preprocessing work by yourself. We focus here on the GAN algorithm itself. The subsequent chapter on customized datasets will introduce in detail how to load your own datasets. Here the processed dataset is obtained directly through the pre-written make_anime_dataset function.

```
    # Dataset path. URL: https://pan.baidu.com/s/1eSifHcA  Access code: g5qa
    img_path = glob.glob(r'C:\Users\z390\Downloads\faces\*.jpg')
    # Create dataset object, return Dataset class and size
    dataset, img_shape, _ = make_anime_dataset(img_path, batch_size,
resize=64)
```

The dataset object is an instance of the tf.data.Dataset class. Operations such as random dispersal, preprocessing, and batching have been completed, and sample batches can be obtained directly, and img_shape is the preprocessed image size.

13.3.2 Generator

The generator network G is formed by stacking 5 transposed convolutional layers in order to realize the layer-by-layer enlargement of the height and width of the feature map, and the layer-by-layer reduction of the number of feature map channels. First, the hidden vector $\boldsymbol{z}$ with a length of 100 is adjusted to a 4-dimensional tensor of $[b,1,1,100]$ through the Reshape operation, and the convolutional layer is transposed in order to enlarge the height and width dimensions, reduce the number of channels, and finally get the color picture with a width of 64 and a channel number of 3. A BN layer is inserted between each convolutional layer to improve training stability, and the convolutional layer chooses not to use a bias vector. The generator class code is implemented as follows:

```
class Generator(keras.Model):
    # Generator class
    def __init__(self):
        super(Generator, self).__init__()
        filter = 64
        # Transposed convolutional layer 1, output channel is filter*8,
kernel is 4, stride is 1, no padding, no bias.
```

```
        self.conv1 = layers.Conv2DTranspose(filter*8, 4,1, 'valid', use_bias
=False)
        self.bn1 = layers.BatchNormalization()
        # Transposed convolutional layer 2
        self.conv2 = layers.Conv2DTranspose(filter*4, 4,2, 'same', use_bias=
False)
        self.bn2 = layers.BatchNormalization()
        # Transposed convolutional layer 3
        self.conv3 = layers.Conv2DTranspose(filter*2, 4,2, 'same', use_bias=
False)
        self.bn3 = layers.BatchNormalization()
        # Transposed convolutional layer 4
        self.conv4 = layers.Conv2DTranspose(filter*1, 4,2, 'same', use_bias=
False)
        self.bn4 = layers.BatchNormalization()
        # Transposed convolutional layer 5
        self.conv5 = layers.Conv2DTranspose(3, 4,2, 'same', use_bias=False)
```

The forward propagation of generator network G is implemented as follow：

```
    def call(self, inputs, training=None):
        x = inputs # [z, 100]
        # Reshape to 4D tensor:(b, 1, 1, 100)
        x = tf.reshape(x, (x.shape[0], 1, 1, x.shape[1]))
        x = tf.nn.relu(x) # activation function
        # Transposed convolutional layer-BN-activation
function:(b, 4, 4, 512)
        x = tf.nn.relu(self.bn1(self.conv1(x), training=training))
        # Transposed convolutional layer-BN-activation
function:(b, 8, 8, 256)
        x = tf.nn.relu(self.bn2(self.conv2(x), training=training))
        # Transposed convolutional layer-BN-activation
function:(b, 16, 16, 128)
        x = tf.nn.relu(self.bn3(self.conv3(x), training=training))
        # Transposed convolutional layer-BN-activation
function:(b, 32, 32, 64)
        x = tf.nn.relu(self.bn4(self.conv4(x), training=training))
        # Transposed convolutional layer-BN-activation
function:(b, 64, 64, 3)
        x = self.conv5(x)
        x = tf.tanh(x) # output x range -1~1

        return x
```

The output size of the generated network is $[b, 64,64,3]$, and the value range is $-1\sim1$.

13.3.3 Discriminator

The discriminator network D is the same as the ordinary classification network. It accepts image tensors of size [b,64,64,3], and continuously extracts features through 5 convolutional layers. The final output size of the convolutional layer is [b ,2,2,1024], and then convert the feature size to [b,1024] through the pooling layer GlobalAveragePooling2D, and finally obtain the probability of the binary classification task through a fully connected layer. The code for the discriminator network class D is implemented as follows:

```
class Discriminator(keras.Model):
    # Discriminator class
    def __init__(self):
        super(Discriminator, self).__init__()
        filter = 64
        # Convolutional layer 1
        self.conv1 = layers.Conv2D(filter, 4, 2, 'valid', use_bias=False)
        self.bn1 = layers.BatchNormalization()
        # Convolutional layer 2
        self.conv2 = layers.Conv2D(filter*2, 4, 2, 'valid', use_bias=False)
        self.bn2 = layers.BatchNormalization()
        # Convolutional layer 3
        self.conv3 = layers.Conv2D(filter*4, 4, 2, 'valid', use_bias=False)
        self.bn3 = layers.BatchNormalization()
        # Convolutional layer 4
        self.conv4 = layers.Conv2D(filter*8, 3, 1, 'valid', use_bias=False)
        self.bn4 = layers.BatchNormalization()
        # Convolutional layer 5
        self.conv5 = layers.Conv2D(filter*16, 3, 1, 'valid', use_bias=False)
        self.bn5 = layers.BatchNormalization()
        # Global pooling layer
        self.pool = layers.GlobalAveragePooling2D()
        # Flatten feature layer
        self.flatten = layers.Flatten()
        # Binary classification layer
        self.fc = layers.Dense(1)
```

The forward calculation process of the discriminator D is implemented as follows:

```
    def call(self, inputs, training=None):
        # Convolutional layer-BN-activation function:(4, 31, 31, 64)
        x = tf.nn.leaky_relu(self.bn1(self.conv1(inputs), training=training)
)
        # Convolutional layer-BN-activation function:(4, 14, 14, 128)
        x = tf.nn.leaky_relu(self.bn2(self.conv2(x), training=training))
        # Convolutional layer-BN-activation function:(4, 6, 6, 256)
        x = tf.nn.leaky_relu(self.bn3(self.conv3(x), training=training))
```

```
        # Convolutional layer-BN-activation function:(4, 4, 4, 512)
        x = tf.nn.leaky_relu(self.bn4(self.conv4(x), training=training))
        # Convolutional layer-BN-activation function:(4, 2, 2, 1024)
        x = tf.nn.leaky_relu(self.bn5(self.conv5(x), training=training))
        # Convolutional layer-BN-activation function:(4, 1024)
        x = self.pool(x)
        # Faltten
        x = self.flatten(x)
        # Output, [b, 1024] => [b, 1]
        logits = self.fc(x)

        return logits
```

The output size of the discriminator is [b,1]. The Sigmoid activation function is not used inside the class, and the probability that b samples belong to the real samples can be obtained through the Sigmoid activation function.

13.3.4 Training and visualization

Discriminator According to formula (13-1), the goal of the discriminator network is to maximize the function $\mathcal{L}(D,G)$, so that the probability of true sample prediction is close to 1, and the probability of generated sample prediction is close to 0. We implement the error function of the discriminator in the d_loss_fn function, label all real samples as 1, and label all generated samples as 0, and maximize the function L(D,G) by minimizing the corresponding cross-entropy loss function. The d_loss_fn function is implemented as follows:

```
def d_loss_fn(generator, discriminator, batch_z, batch_x, is_training):
    # Loss function for discriminator
    # Generate images from generator
    fake_image = generator(batch_z, is_training)
    # Distinguish images
    d_fake_logits = discriminator(fake_image, is_training)
    # Determine whether the image is real or not
    d_real_logits = discriminator(batch_x, is_training)
    # The error between real image and 1
    d_loss_real = celoss_ones(d_real_logits)
    # The error between generated image and 0
    d_loss_fake = celoss_zeros(d_fake_logits)
    # Combine loss
    loss = d_loss_fake + d_loss_real

    return loss
```

The celoss_ones function calculates the cross entropy loss between the current predicted probability and label 1. The code is as follows:

```
def celoss_ones(logits):
    # Calculate the cross entropy belonging to and label 1
```

```
    y = tf.ones_like(logits)
    loss = keras.losses.binary_crossentropy(y, logits, from_logits=True)
    return tf.reduce_mean(loss)
```

The celoss_zeros function calculates the cross entropy loss between the current predicted probability and label 0. The code is as follows:

```
def celoss_zeros(logits):
    # Calculate the cross entropy that belongs to and the note is 0
    y = tf.zeros_like(logits)
    loss = keras.losses.binary_crossentropy(y, logits, from_logits=True)
    return tf.reduce_mean(loss)
```

Generator The training goal of generator network is to minimize the $\mathcal{L}(D,G)$ objective function. Since the real sample has nothing to do with the generator, the error function only needs to minimize $\mathbb{E}_{\boldsymbol{z}\sim p_z(\cdot)}\log\left(1-D_\theta\left(G_\phi(\boldsymbol{z})\right)\right)$. The cross entropy error at this time can be minimized by marking the generated sample as 1. It should be noted that in the process of backpropagating errors, the discriminator also participates in the construction of the calculation graph, but at this stage only the generator network parameters need to be updated. The error function of the generator is as follows:

```
def g_loss_fn(generator, discriminator, batch_z, is_training):
    # Generate images
    fake_image = generator(batch_z, is_training)
    #  When training the generator network, it is necessary to force the
generated image to be judged as true
    d_fake_logits = discriminator(fake_image, is_training)
    # Calculate error between generated images and 1
    loss = celoss_ones(d_fake_logits)

    return loss
```

Network training In each Epoch, first randomly sample the hidden vector from the prior distribution $p_z(\cdot)$, randomly sample the real pictures from the true data set, calculate the loss of the discriminator network through the generator and the discriminator, and optimize the discriminator network parameters θ. When training the generator, the discriminator is needed to calculate the error, but only the gradient information of the generator is calculated and ϕ is updated. Here set the discriminator training times $k=5$, and set the generator training time as one.

First, create the generator network and the discriminator network, and create the corresponding optimizers respectively as below:

```
    generator = Generator() #  Create generator
    generator.build(input_shape = (4, z_dim))
    discriminator = Discriminator() #  Create discriminator
```

```
    discriminator.build(input_shape=(4, 64, 64, 3))
    # Create optimizers for generator and discriminator respectively
    g_optimizer = keras.optimizers.Adam(learning_rate=learning_rate, beta_1=
0.5)
    d_optimizer = keras.optimizers.Adam(learning_rate=learning_rate, beta_1=
0.5)
```

The main training part of the code is implemented as follows:

```
    for epoch in range(epochs): #  Train epochs times
        # 1. Train discriminator
        for _ in range(5):
            # Sample hidden vectors
            batch_z = tf.random.normal([batch_size, z_dim])
            batch_x = next(db_iter) # Sample real images
            # Forward calculation - discriminator
            with tf.GradientTape() as tape:
                d_loss = d_loss_fn(generator, discriminator, batch_z, batch_
x, is_training)
            grads = tape.gradient(d_loss, discriminator.trainable_variables)
            d_optimizer.apply_gradients(zip(grads, discriminator.trainable_v
ariables))
        # 2. Train generator
        # Sample hidden vectors
        batch_z = tf.random.normal([batch_size, z_dim])
        batch_x = next(db_iter) # Sample real images
        # Forward calculation - generator
        with tf.GradientTape() as tape:
            g_loss = g_loss_fn(generator, discriminator, batch_z, is_trainin
g)
        grads = tape.gradient(g_loss, generator.trainable_variables)
        g_optimizer.apply_gradients(zip(grads, generator.trainable_variables
))
```

Every 100 Epochs, a picture generation test is performed. The hidden vector is randomly sampled from the prior distribution, sent to the generator to obtain the generated picture which is saved as a file.

As shown in Figure 13.8, it shows a sample of generated pictures saved by the DCGAN model during the training process. It can be observed that most of the pictures have clear subjects, vivid colors, rich picture diversity, and the generated pictures are close to the real pictures in the data set. At the same time, it can be found that a small amount of generated pictures are still damaged, and the main body of the pictures cannot be recognized by human eyes. To obtain the image generation effect shown in Figure 13.8, it is necessary to carefully design the network model structure and fine-tune the network hyperparameters.

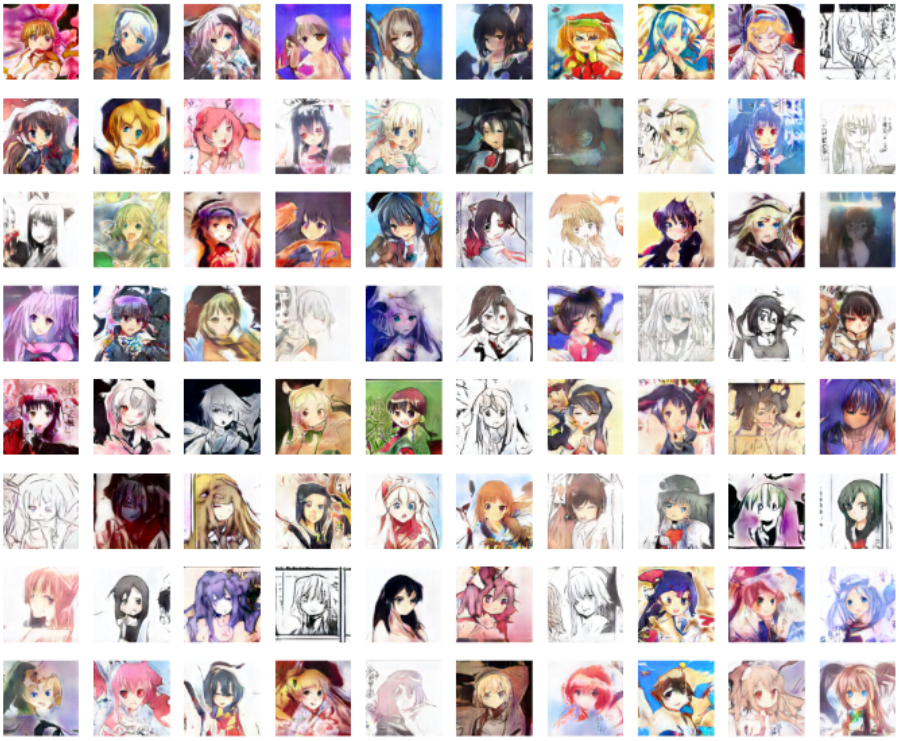


Figure 0.8 DCGAN image generation effect

13.4 GAN variants

In the original GAN paper, Ian Goodfellow analyzed the convergence of the GAN network from a theoretical level, and tested the effect of image generation on multiple classic image data sets, as shown in Figure 13.9, where Figure 13.9 (a) is the MNIST dataset, Figure 13.9 (b) is the Toronto Face dataset, Figure 13.9 (c) and Figure 13.9 (d) are the CIFAR10 dataset.

Figure 0.9 Original GAN image generation effect [1]

It can be seen that the original GAN model is not outstanding in terms of image generation effect, and the difference from VAE is not obvious. At this time, it does not show its powerful distribution approximation ability. However, because GAN is relatively new in theory, there are

many areas for improvement, which greatly stimulated the research interest of the academic community. In the next few years, GAN research is in full swing, and substantial progress has also been made. Next we will introduce several significant GAN variants.

13.4.1 DCGAN

The initial GAN network is mainly based on the fully connected layer to realize the generator G and the discriminator D. Due to the high dimensionality of the picture and the huge amount of network parameters, the training effect is not excellent. DCGAN [2] proposed a generator network implemented using transposed convolutional layers, and a discriminator network implemented by ordinary convolutional layers, which greatly reduces the amount of network parameters and greatly improves the effect of image generation, showing that the GAN model has the potential of outperforming the VAE model in image generation. In addition, the author of DCGAN also proposed a series of empirical GAN network training techniques, which were proved to be beneficial to the stable training of the GAN network. We have used the DCGAN model to complete the actual picture generation of the animation avatars.

13.4.2 InfoGAN

InfoGAN [3] tried to use an unsupervised way to learn the Interpretable Representation of the interpretable hidden vector $\boldsymbol{z}$ of the input $\boldsymbol{x}$, that is, it is hoped that the hidden vector $\boldsymbol{z}$ can correspond to the semantic features of the data. For example, for MNIST handwritten digital pictures, we can consider the category, font size, and writing style of the digits to be hidden variables of the picture. We hope that the model can learn these disentangled interpretable feature representation methods, so that the hidden variables can be controlled artificially to generate a sample of the specified content. For the CelebA celebrity photo dataset, it is hoped that the model can separate features such as hairstyles, glasses wearing conditions, facial expressions, etc., to generate face images of specified shapes.

What are the benefits of disentangled interpretable features? It can make the neural network more interpretable. For example, $\boldsymbol{z}$ contains some separate interpretable features, then we can obtain generated data with different semantics by only changing the features at this position. As shown in Figure 13.10, subtracting the hidden vectors of "men with glasses" and "men without glasses" and adding them to the hidden vectors of "women without glasses" can generate a picture of "women with glasses".

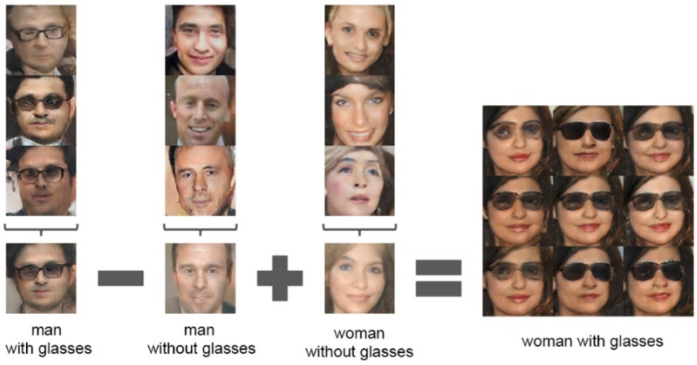


Figure 0.10 Schematic diagram of separated features [3]

13.4.3 CycleGAN

CycleGAN [4] is an unsupervised algorithm for image style conversion proposed by Zhu Junyan. Because the algorithm is clear and simple, and the results are better, this work has received a lot of praise. The basic assumption of CycleGAN is that if you switch from picture A to picture B, and then from picture B to A', then A' should be the same picture as A. Therefore, in addition to setting up the standard GAN loss item, CycleGAN also adds Cycle Consistency Loss to ensure that A' is as close to A as possible. The conversion effect of CycleGAN pictures is shown in Figure 13.11.

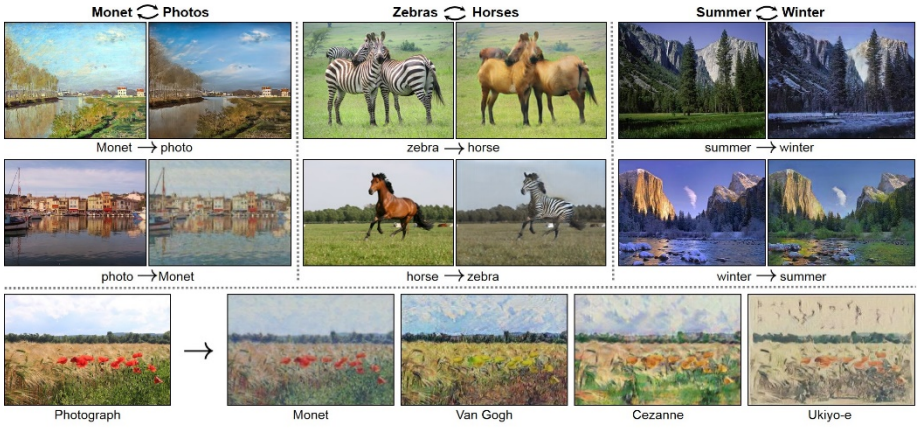


Figure 0.11 Image conversion effect [4]

13.4.4 WGAN

The training problem of GAN has been criticized all the time, and it is prone to the phenomenon of training non-convergence and mode collapse. WGAN [5] analyzed the flaws of the original GAN using JS divergence from a theoretical level, and proposed that the Wasserstein distance can be used to solve this problem. In WGAN-GP [6], the author proposed that by adding a gradient penalty term, the WGAN algorithm was well realized from the engineering level, and the advantages of WGAN training stability were confirmed.

13.4.5 Equal GAN

From the birth of GAN to the end of 2017, GAN Zoo has collected more than 214 GAN network variants. These GAN variants have more or less proposed some innovations, but several researchers from Google Brain provided another point in a paper [7]: There is no evidence that the GAN variant algorithms we tested have been consistently better than the original GAN paper. In that paper, these GAN variants are compared fairly and comprehensively. With sufficient computing resources, it is found that almost all GAN variants can achieve similar performance (FID score). This work reminds the industry whether these GAN variants are essentially innovative.

13.4.6 Self-Attention GAN

The Attention mechanism has been widely used in natural language processing (NLP). Self-Attention GAN (SAGAN) [8] borrowed from the Attention mechanism and proposed a variant of GAN based on the self-attention mechanism. SAGAN improved the fidelity index of the picture: Inception score from the 36.8 to 52.52, and Frechet Inception distance from 27.62 to 18.65. From the effect of image generation perspective, SAGAN's breakthrough is very significant, and it also inspired the industry's attention to the self-attention mechanism.

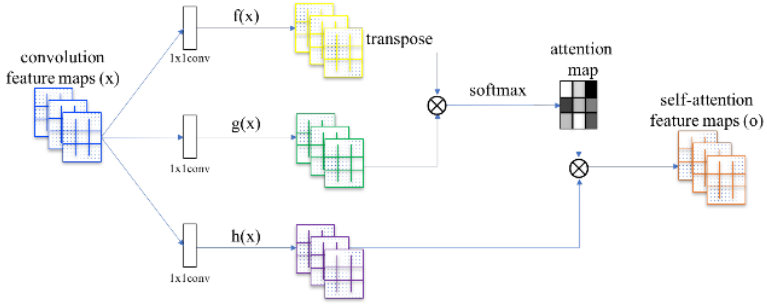


Figure 0.12 Attention mechanism in SAGAN [8]

13.4.7 BigGAN

On the basis of SAGAN, BigGAN [9] attempts to extend the training of GAN to a large scale, using techniques such as orthogonal regularization to ensure the stability of the training process. The significance of BigGAN is to inspire people that the training of GAN networks can also benefit from big data and large computing power. The effect of BigGAN image generation has reached an unprecedented height: the Inception score record has increased to 166.5 (an increase of 52.52); Frechet Inception Distance has dropped to 7.4, which has been reduced by 18.65. As shown in Figure 13.13, the image resolution can reach 512×512, and the image details are extremely realistic.

Figure 0.13 BigGAN generated images

13.5 Nash Equilibrium

Now we analyze from the theoretical level, through the training method of game learning, what equilibrium state the generator G and the discriminator D will reach. Specifically, we will explore the following two questions:

- Fix G, what optimal state D^* will D converge to?
- After D reaches the optimal state D^*, what state will G converge to?

First, we give an intuitive explanation through the example of one-dimensional normal distribution $\boldsymbol{x}_r \sim p_r(\cdot)$. As shown in Figure 13.14, the black dashed curve represents the real data distribution $p_r(\cdot)$, which is a normal distribution $\mathcal{N}(\mu, \sigma^2)$, and the green solid line represents the distribution $\boldsymbol{x}_f \sim p_g(\cdot)$ learned by the generator network. The blue dotted line represents the decision boundary curve of the discriminator. Figure 13.14 (a), (b), (c), and (d) represent the learning trajectory of the generator network, respectively. In the initial state, as shown in Figure 13.14(a), the distribution of $p_g(\cdot)$ is quite different from $p_r(\cdot)$, and the discriminator can easily learn a clear decision boundary, which is the blue dotted line in Figure 13.14(a), which sets the sampling point from $p_g(\cdot)$ as 0 and the sampling point in $p_r(\cdot)$ as 1. As the distribution $p_g(\cdot)$ of the generator network approaches the true distribution $p_r(\cdot)$, it becomes more and more difficult for the discriminator to distinguish between true and false samples, as shown in Figures 13.14(b)(c). Finally, when the distribution $p_g(\cdot) = p_r(\cdot)$ learned by the generator network, the samples extracted from the generator network are very realistic, and the discriminator cannot distinguish the difference, that is, the probability of determining the true and false samples is equal, as shown in Figure 13.14(d).

This example intuitively explains the training process of the GAN network.

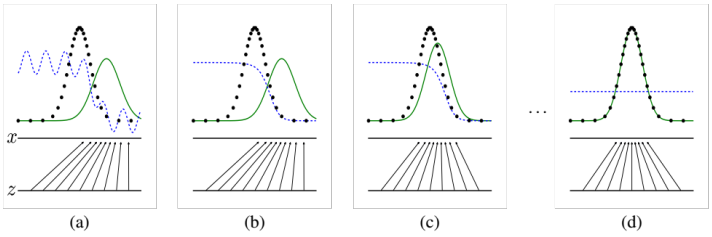


Figure 0.14 Nash Equilibrium [1]

13.5.1 Discriminator State

Now let's derive the first question. Review the loss function of GAN:

$$\mathcal{L}(G,D) = \int_{\boldsymbol{x}} p_r(\boldsymbol{x}) \log\big(D(\boldsymbol{x})\big)\, d\boldsymbol{x} + \int_{\boldsymbol{z}} p_z(\boldsymbol{z}) \log\Big(1 - D\big(g(\boldsymbol{z})\big)\Big)\, d\boldsymbol{z}$$

$$= \int_{x} p_r(\boldsymbol{x}) \log\big(D(\boldsymbol{x})\big) + p_g(\boldsymbol{x}) \log\big(1 - D(\boldsymbol{x})\big)\, d\boldsymbol{x}$$

For the discriminator D, the optimization goal is to maximize the $\mathcal{L}(G,D)$ function, and the maximum value of the following function needs to be found:

$$f_\theta = p_r(\boldsymbol{x}) \log\big(D(\boldsymbol{x})\big) + p_g(\boldsymbol{x}) \log\big(1 - D(\boldsymbol{x})\big)$$

where θ is the network parameter of the discriminator D.

Let us consider the maximum value of the more general function of f_θ:

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

The maximum value of the function $f(x)$ is required. Consider the derivative of $f(x)$:

$$\frac{\mathrm{d}f(x)}{\mathrm{d}x} = A\frac{1}{\ln 10}\frac{1}{x} - B\frac{1}{\ln 10}\frac{1}{1-x}$$

$$= \frac{1}{\ln 10}\left(\frac{A}{x} - \frac{B}{1-x}\right)$$

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

Let $\frac{\mathrm{d}f(x)}{\mathrm{d}x} = 0$, we can find the extreme points of the $f(x)$ function:

$$x = \frac{A}{A+B}$$

Therefore, it can be known that the extreme points of the f_θ function are also:

$$D_\theta = \frac{p_r(\boldsymbol{x})}{p_r(\boldsymbol{x}) + p_g(\boldsymbol{x})}$$

That is to say, when the discriminator network D_θ is in the D_{θ^*} state, the f_θ function takes the maximum value, and the $\mathcal{L}(G,D)$ function also takes the maximum value.

Now back to the problem of maximizing $\mathcal{L}(G,D)$, the maximum point of $\mathcal{L}(G,D)$ is obtained at:

$$D^* = \frac{A}{A+B} = \frac{p_r(\boldsymbol{x})}{p_r(\boldsymbol{x}) + p_g(\boldsymbol{x})}$$

which is also the optimal state D^* of D_θ.

13.5.2 Generator State

Before deriving the second question, we first introduce another distribution distance metric similar to KL divergence: JS divergence, which is defined as a combination of KL divergence:

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

$$D_{JS}(p||q) = \frac{1}{2}D_{KL}\left(p||\frac{p+q}{2}\right) + \frac{1}{2}D_{KL}\left(q||\frac{p+q}{2}\right)$$

JS divergence overcomes the asymmetry of KL divergence.

When D reaches the optimal state D^*, let us consider the JS divergence of p_r and p_g at this time:

$$D_{JS}(p_r||p_g) = \frac{1}{2}D_{KL}\left(p_r||\frac{p_r+p_g}{2}\right) + \frac{1}{2}D_{KL}\left(p_g||\frac{p_r+p_g}{2}\right)$$

According to the definition of KL divergence:

$$D_{JS}(p_r||p_g) = \frac{1}{2}\left(\log 2 + \int_{\boldsymbol{x}} p_r(\boldsymbol{x}) \log \frac{p_r(\boldsymbol{x})}{p_r + p_g(\boldsymbol{x})} d\boldsymbol{x}\right)$$
$$+\frac{1}{2}\left(\log 2 + \int_{\boldsymbol{x}} p_g(\boldsymbol{x}) \log \frac{p_g(\boldsymbol{x})}{p_r + p_g(\boldsymbol{x})} d\boldsymbol{x}\right)$$

Combining the constant terms, we can get:

$$D_{JS}(p_r||p_g) = \frac{1}{2}(\log 2 + \log 2)$$
$$+\frac{1}{2}\left(\int_{\boldsymbol{x}} p_r(\boldsymbol{x}) \log \frac{p_r(\boldsymbol{x})}{p_r + p_g(\boldsymbol{x})} d\boldsymbol{x} + \int_{\boldsymbol{x}} p_g(\boldsymbol{x}) \log \frac{p_g(\boldsymbol{x})}{p_r + p_g(\boldsymbol{x})} d\boldsymbol{x}\right)$$

i.e.:

$$D_{JS}(p_r||p_g) = \frac{1}{2}(\log 4)$$
$$+\frac{1}{2}\left(\int_{\boldsymbol{x}} p_r(\boldsymbol{x}) \log \frac{p_r(\boldsymbol{x})}{p_r + p_g(\boldsymbol{x})} d\boldsymbol{x} + \int_{\boldsymbol{x}} p_g(\boldsymbol{x}) \log \frac{p_g(\boldsymbol{x})}{p_r + p_g(\boldsymbol{x})} d\boldsymbol{x}\right)$$

Consider when the network reaches D^*, the loss function at this time is:

$$\mathcal{L}(G, D^*) = \int_{x} p_r(\boldsymbol{x}) \log\big(D^*(\boldsymbol{x})\big) + p_g(\boldsymbol{x}) \log\big(1 - D^*(\boldsymbol{x})\big)\, d\boldsymbol{x}$$
$$= \int_{\boldsymbol{x}} p_r(\boldsymbol{x}) \log \frac{p_r(\boldsymbol{x})}{p_r + p_g(\boldsymbol{x})} d\boldsymbol{x} + \int_{\boldsymbol{x}} p_g(\boldsymbol{x}) \log \frac{p_g(\boldsymbol{x})}{p_r + p_g(\boldsymbol{x})} d\boldsymbol{x}$$

Therefore, when the discriminator network reaches D^*, $D_{JS}(p_r||p_g)$ and $\mathcal{L}(G, D^*)$ satisfy the relationship:

$$D_{JS}(p_r||p_g) = \frac{1}{2}\big(\log 4 + \mathcal{L}(G, D^*)\big)$$

i.e.:

$$\mathcal{L}(G, D^*) = 2D_{JS}(p_r||p_g) - 2\log 2$$

For the generator network G, the training target is $\min_G \mathcal{L}(G, D)$, considering the nature of the JS divergence:

$$D_{JS}(p_r||p_g) \geq 0$$

Therefore, $\mathcal{L}(G, D^*)$ obtains the minimum value only when $D_{JS}(p_r||p_g) = 0$ (at this time $p_g = p_r$), $\mathcal{L}(G, D^*)$ obtains the minimum value:

$$\mathcal{L}(G^*, D^*) = -2\,log\,2$$

At this time, the state of the generator network G^* is:

$$p_g = p_r$$

That is, the learned distribution p_g of G^* is consistent with the real distribution p_r, and the network reaches a balance point. At this time:

$$D^* = \frac{p_r(\boldsymbol{x})}{p_r(\boldsymbol{x}) + p_g(\boldsymbol{x})} = 0.5$$

13.5.3 Nash Equilibrium Point

Through the above derivation, we can conclude that the generation network G will eventually converge to the true distribution, namely:

$$p_g = p_r$$

At this time, the generated sample and the real sample come from the same distribution, and it is difficult to distinguish between true and false. The discriminator has the same probability to judge as true or false, that is

$$D(\cdot) = 0.5$$

At this time, the loss function is

$$\mathcal{L}(G^*, D^*) = -2\log 2$$

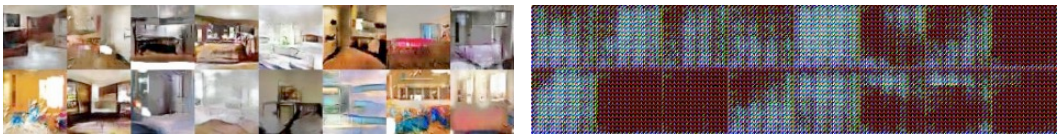
13.6 GAN training difficulty

Although the GAN network can learn the true distribution of data from the theoretical level, the problem of difficulty in GAN network training often arises in engineering implementation, which is mainly reflected in that the GAN model is more sensitive to hyperparameters, and it is necessary to carefully select the hyperparameters that can make the model work. Hyperparameter settings are also prone to mode collapse.

13.6.1 Hyperparameter sensitivity

Hyper-parameter sensitivity means that the network's structure setting, learning rate, initialization state and other hyper-parameters have a greater impact on the training process of the network. A small amount of hyper-parameter adjustment may lead to completely different network training results. As shown in Figure 13.15, Figure (a) shows the generated samples obtained from good training of the GAN model. The network in Figure (b) does not use the Batch Normalization layer and other settings, resulting in unstable GAN network training and failure to converge. The generated samples are different from each other. The real sample gap is very large.

In order to train the GAN network well, the author of the DCGAN paper proposes not to use the Pooling layer, not to use the fully connected layer, to use the Batch Normalization layer more, and the activation function in the generated network should use ReLU. The activation function of the last layer should be tanh, and the activation function of the discriminator network should use a series of empirical training techniques such as LeakyLeLU. However, these techniques can only avoid the phenomenon of training instability to a certain extent, and do not explain from the theoretical level why there is training difficulty and how to solve the problem of training instability.



(a) (b)

Figure 0.15 Hyperparameter sensitive example [5]

13.6.2 Model collapse

Mode collapse refers to the phenomenon that the sample generated by the model is single and the diversity is poor. Since the discriminator can only identify whether a single sample is sampled from the true distribution, and does not impose explicit constraints on the sample diversity, the generative model may tend to generate a small number of high-quality samples in a partial interval of the true distribution, without learning all the true distributions. The phenomenon of model collapse is more common in GAN, as shown in Figure 13.16. During the training process, it can be observed by visualizing the samples of the generator network that the types of pictures generated are very single, and the generator network always tends to generate samples of a certain single style to fool the discriminator.

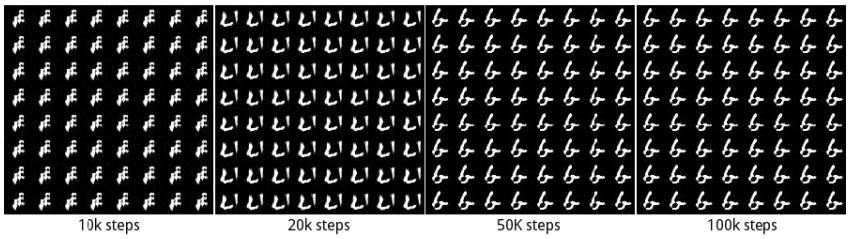


Figure 0.16 Image generation - model collapsed [10]

Another example of intuitive understanding of mode collapse is shown in Figure 13.17. The first row is the training process of the generator network without mode collapse, and the last column is the real distribution, that is, the 2D Gaussian mixture model. The second row shows the training process of generator network with model collapse. The last column is the true distribution. It can be seen that the real distribution is a mixture of 8 Gaussian models. After model collapse occurs, the generator network always tends to approach a narrow interval of the real distribution, as shown in the first 6 columns of the second row in Figure 13.17. The samples from this interval of can often be judged as real samples with a higher probability in the discriminator, thus deceiving the discriminator. But this phenomenon is not what we want to see. We hope that the generator network can approximate the real distribution, rather than a certain part of the real distribution.

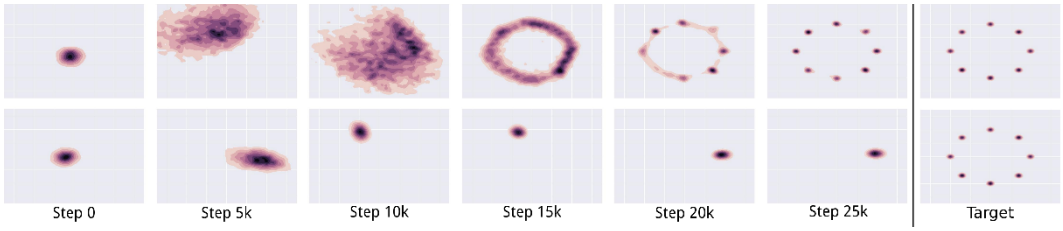


Figure 0.17 Schematic diagram of model collapse [10]

So how to solve the problem of GAN training so that GAN can be trained more stably like ordinary neural networks? The WGAN model provides a solution.

13.7 WGAN principle

The WGAN algorithm analyzes the reasons for the instability of GAN training from a theoretical level, and proposes an effective solution. So what makes GAN training so unstable? WGAN proposed that is because the gradient surface of the JS divergence on the non-overlapping distributions p and q is always 0. As shown in Figure 13.19, when the distributions p and q do not overlap, the gradient value of the JS divergence is always 0, which leads to the gradient vanishing phenomenon, therefore, the parameters cannot be updated for a long time, and the network cannot converge.

Next we will elaborate on the defects of JS divergence and how to solve this defect.

13.7.1 JS divergence disadvantage

In order to avoid too much theoretical derivation, we use a simple distribution example to explain the defects of JS divergence. Consider two distributions p and q that are completely non-overlapping ($\theta \neq 0$), where the distribution p is:

$$\forall(x,y) \in p, x = 0, y \sim \mathrm{U}(0,1)$$

And the distribution of q is：

$$\forall(x,y) \in q, x = \theta, y \sim \mathrm{U}(0,1)$$

where $\theta \in R$, when $\theta = 0$, the distributions p and q overlap, and the two are equal; when $\theta \neq 0$, the distributions p and q do not overlap.

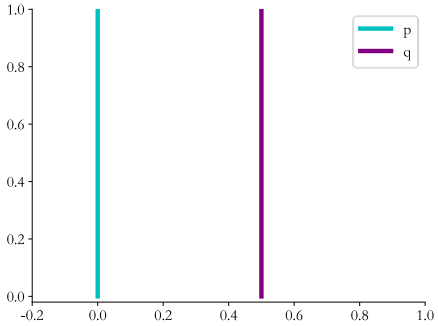


Figure 0.18 Schematic diagram of distribution p and q

Let us analyze the variation of the JS divergence between the above distributions p and q with θ. According to the definition of KL divergence and JS divergence, calculate the JS divergence $D_{JS}(p||q)$ when $\theta = 0$:

$$D_{KL}(p||q) = \sum_{x=0, y\sim \mathrm{U}(0,1)} 1 \cdot \log\frac{1}{0} = +\infty$$

$$D_{KL}(q||p) = \sum_{x=\theta, y\sim U(0,1)} 1 \cdot \log\frac{1}{0} = +\infty$$

$$D_{JS}(p||q) = \frac{1}{2}\left(\sum_{x=0, y\sim U(0,1)} 1 \cdot \log\frac{1}{1/2} + \sum_{x=0, y\sim U(0,1)} 1 \cdot \log\frac{1}{1/2}\right) = \log 2$$

When $\theta = 0$, the two distributions completely overlap. At this time, the JS divergence and KL divergence both achieve the minimum value, which is 0:

$$D_{KL}(p||q) = D_{KL}(q||p) = D_{JS}(p||q) = 0$$

From the above derivation, we can get the trend of $D_{JS}(p||q)$ with θ:

$$D_{JS}(p||q) = \begin{cases} \log 2 & \theta \neq 0 \\ 0 & \theta = 0 \end{cases}$$

In other words, when the two distributions do not overlap at all, regardless of the distance between the distributions, the JS divergence is a constant value $\log 2$, then the JS divergence will not be able to produce effective gradient information. When the two distributions overlap, the JS divergence changes smoothly and produces effective gradient information. When the two distributions completely coincide, the JS divergence takes the minimum value of 0. As shown in Figure 13.19, the red curve divides the two normal distributions. Since the two distributions do not overlap, the gradient value at the generated sample position is always 0, and the parameters of the generator network cannot be updated, resulting in difficulty in network training.

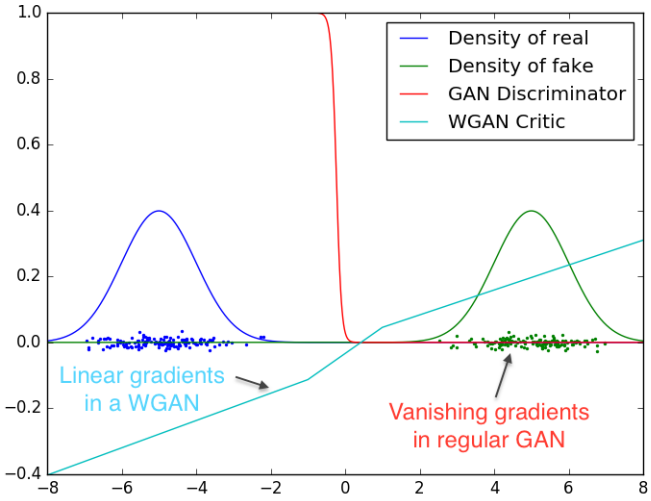


Figure 0.19 Gradient vanishing of JS divergence [5]

Therefore, the JS divergence cannot smoothly measure the distance between the distributions when the distributions p and q do not overlap. As a result, effective gradient information cannot be generated at this position, and the GAN training is unstable. To solve this problem, we need to use a better distribution distance measurement, so that it can smoothly reflect the true distance change between the distributions even when the distributions p and q do not overlap.

13.7.2 EM distance

The WGAN paper found that JS divergence leads to the instability of GAN training, and introduced a new distribution distance measurement method: Wasserstein distance, also called Earth-Mover Distance (EM distance), which represents the minimum cost of transforming a distribution to another distribution. It's defined as:

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

where $\prod(p,q)$ is the set of all possible joint distributions combined by the distributions p and q. For each possible joint distribution $\gamma \sim \prod(p,q)$, calculate the expectation distance $\mathbb{E}_{(x,y)\sim\gamma}[\|x-y\|]$ of $\|x-y\|$, where (x,y) is sampled from the joint distribution γ. Different joint distributions γ have different expectations $\mathbb{E}_{(x,y)\sim\gamma}[\|x-y\|]$, and the infimum of these expectations is defined as the Wasserstein distance of distributions p and q. Where $\inf\{\cdot\}$ represents the infimum of the set, for example, the infimum of $\{x|1<x<3, x\in R\}$ is 1.

Continuing to consider the example in Figure 13.18, we directly give the expression of the EM distance between the distributions p and q:

$$W(p,q) = |\theta|$$

Draw the curves of JS divergence and EM distance, as shown in Figure 13.20. It can be seen that the JS divergence is not continuous at $\theta = 0$, the other position derivatives are all 0, and the EM distance can always produce effective derivative information. Therefore, EM distance is more suitable for guiding the training of GAN network than JS divergence.

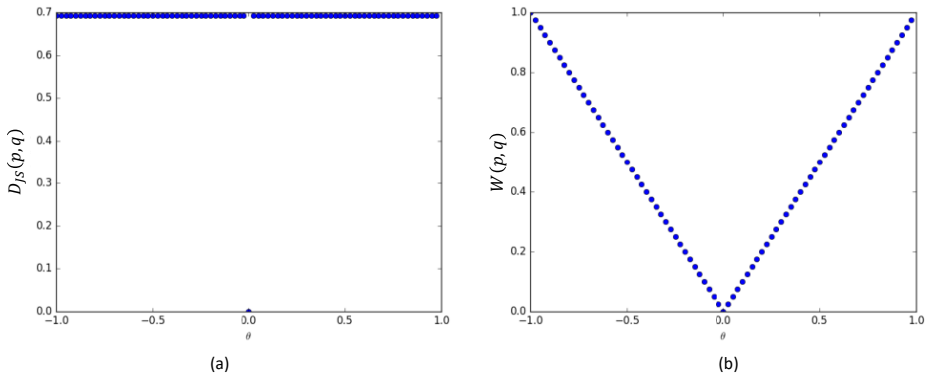


Figure 0.20 JS divergence and EM distance change curve with θ WGAN-GP

Considering that it is almost impossible to traverse all the joint distributions γ to calculate the distance expectation $\mathbb{E}_{(x,y)\sim\gamma}[\|x-y\|]$ of $\|x-y\|$, so it's not realistic to calculate the distance between the distribution p_g of the generator network and $W(p_r,p_g)$. Based on the Kantorovich-Rubinstein duality, the WGAN author converts the direct calculation of $W(p_r,p_g)$ into:

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

where $\sup\{\cdot\}$ represents the supremum of the set, $||f||_L \le K$ represents the function $f: R \to R$

which satisfies the K-order Lipschitz continuity, that is,

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

Therefore, we use the discriminant network $D_\theta(\boldsymbol{x})$ to parameterize the $f(\boldsymbol{x})$ function, under the condition that D_θ satisfies the 1-Lipschitz constraint, that is, $K = 1$, at this time:

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

Therefore, the problem of solving $W(p_r, p_g)$ can be transformed into:

$$\max_\theta \mathbb{E}_{\boldsymbol{x}\sim p_r}[D_\theta(\boldsymbol{x})] - \mathbb{E}_{\boldsymbol{x}\sim p_g}[D_\theta(\boldsymbol{x})]$$

This is the optimization goal of the discriminator D. The discriminant network function $D_\theta(\boldsymbol{x})$ needs to satisfy the 1-Lipschitz constraint:

$$\nabla_{\hat{\boldsymbol{x}}} D(\hat{\boldsymbol{x}}) \leq I$$

In the WGAN-GP paper, the author proposes to increase the gradient penalty method to force the discriminator network to meet the first-order-Lipschitz function constraint, and the author found that the engineering effect is better when the gradient value is constrained around 1, so the gradient penalty term is defined as:

$$\text{GP} \triangleq \mathbb{E}_{\hat{\boldsymbol{x}}\sim P_{\hat{\boldsymbol{x}}}}[(\|\nabla_{\hat{\boldsymbol{x}}} D(\hat{\boldsymbol{x}})\|_2 - 1)^2]$$

Therefore, the training objective of WGAN discriminator D is:

$$\max_\theta \mathcal{L}(G, D) = \underbrace{\mathbb{E}_{\boldsymbol{x}_r\sim p_r}[D(\boldsymbol{x}_r)] - \mathbb{E}_{\boldsymbol{x}_f\sim p_g}[D(\boldsymbol{x}_f)]}_{\text{EM 距离}} - \underbrace{\lambda \mathbb{E}_{\hat{\boldsymbol{x}}\sim P_{\hat{\boldsymbol{x}}}}[(\|\nabla_{\hat{\boldsymbol{x}}} D(\hat{\boldsymbol{x}})\|_2 - 1)^2]}_{\text{GP 惩罚项}}$$

where $\hat{\boldsymbol{x}}$ comes from the linear difference between $\boldsymbol{x}_r$ and $\boldsymbol{x}_f$:

$$\hat{x} = t\boldsymbol{x}_r + (1-t)\boldsymbol{x}_f, t \in [0,1]$$

The goal of the discriminator D is to minimize the above-mentioned error $\mathcal{L}(G, D)$, that is, to force the EM distance $\mathbb{E}_{\boldsymbol{x}_r\sim p_r}[D(\boldsymbol{x}_r)] - \mathbb{E}_{\boldsymbol{x}_f\sim p_g}[D(\boldsymbol{x}_f)]$ as large as possible, and $\|\nabla_{\hat{\boldsymbol{x}}} D(\hat{\boldsymbol{x}})\|_2$ close to 1.

The training objectives of WGAN generator G are:

$$\min_\phi \mathcal{L}(G, D) = \underbrace{\mathbb{E}_{\boldsymbol{x}_r\sim p_r}[D(\boldsymbol{x}_r)] - \mathbb{E}_{\boldsymbol{x}_f\sim p_g}[D(\boldsymbol{x}_f)]}_{\text{EM distance}}$$

That is, the EM distance between the generator's distribution p_g and the real distribution p_r is as small as possible. Considering that $\mathbb{E}_{\boldsymbol{x}_r\sim p_r}[D(\boldsymbol{x}_r)]$ has nothing to do with the generator, the training objective of the generator is abbreviated as:

$$\min_\phi \mathcal{L}(G, D) = -\mathbb{E}_{\boldsymbol{x}_f\sim p_g}[D(\boldsymbol{x}_f)]$$

$$= -\mathbb{E}_{\boldsymbol{z}\sim p_z(\cdot)}[D(G(\boldsymbol{z}))]$$

From the implementation point of view, the output of the discriminator network D does not need to add a Sigmoid activation function. This is because the original version of the discriminator is a binary classification network, the Sigmoid function is added to obtain the probability of belonging to a certain category; while the discriminator in WGAN is used to measure the EM

distance between the distribution p_g of the generator network and the real distribution p_r. It belongs to the real number space, so there is no need to add a Sigmoid activation function. When calculating the error function, WGAN also does not have a log function. When training WGAN, WGAN authors recommend using RMSProp or SGD and other optimizers without momentum.

WGAN discovered the reason why the original GAN is prone to training instability from the theoretical level, and gave a new distance metric and engineering implementation solution, which achieved good results. WGAN also alleviates the problem of model collapse to a certain extent, and the model using WGAN is not prone to model collapse. It should be noted that WGAN generally does not improve the generation effect of the model, but only ensures the stability of model training. Of course, the training stably is also a prerequisite for good model performance. As shown in Figure 13.21, the original version of DCGAN showed unstable training when the BN layer and other settings were not used. Under the same settings, using WGAN to train the discriminator can avoid this phenomenon, as shown in Figure 13.22.

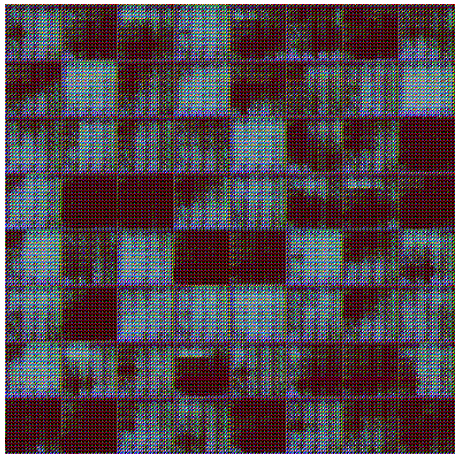


Figure 0.21 DCGAN generator effect without BN layer [5]

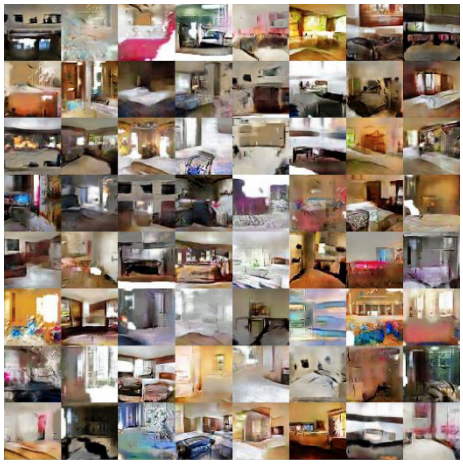


Figure 0.22 WGAN generator effect without BN layer [5]

13.8 Hands-on WGAN-GP

The WGAN-GP model can be modified slightly on the basis of the original GAN

implementation. The output of the discriminator D of the WGAN-GP model is no longer the probability of the sample category, and the output does not need to add the Sigmoid activation function. At the same time, we need to add a gradient penalty term as follow:

```
def gradient_penalty(discriminator, batch_x, fake_image):
    # Gradient penalty term calculation function
    batchsz = batch_x.shape[0]

    # Each sample is randomly sampled at t for interpolation
    t = tf.random.uniform([batchsz, 1, 1, 1])
    # Automatically expand to the shape of x, [b, 1, 1, 1] => [b, h, w, c]
    t = tf.broadcast_to(t, batch_x.shape)
    # Perform linear interpolation between true and false pictures
    interplate = t * batch_x + (1 - t) * fake_image
    # Calculate the gradient of D to interpolated samples in a gradient
environment
    with tf.GradientTape() as tape:
        tape.watch([interplate]) # Add to the gradient watch list
        d_interplote_logits = discriminator(interplate)
    grads = tape.gradient(d_interplote_logits, interplate)

    # Calculate the norm of the gradient of each
sample:[b, h, w, c] => [b, -1]
    grads = tf.reshape(grads, [grads.shape[0], -1])
    gp = tf.norm(grads, axis=1) #[b]
    # Calculate the gradient penalty
    gp = tf.reduce_mean( (gp-1.)**2 )

    return gp
```

The loss function calculation of WGAN discriminator is different from GAN. WGAN directly maximizes the output value of real samples and minimizes the output value of generated samples. There is no cross-entropy calculation process. The code is implemented as follows:

```
def d_loss_fn(generator, discriminator, batch_z, batch_x, is_training):
    # Calculate loss function for D
    fake_image = generator(batch_z, is_training) # Generated sample
    d_fake_logits = discriminator(fake_image, is_training) # Output of
generated sample
    d_real_logits = discriminator(batch_x, is_training) # Output of real
sample
    # Calculate gradient penalty term
    gp = gradient_penalty(discriminator, batch_x, fake_image)
    # WGAN-GP loss function of D. Here is not to calculate the cross
entropy, but to directly maximize the output of the positive sample
    # Minimize the output of false samples and the gradient penalty term
```

```
    loss = tf.reduce_mean(d_fake_logits) - tf.reduce_mean(d_real_logits) + 1
0. * gp

    return loss, gp
```

The loss function of the WGAN generator G only needs to maximize the output value of the generated sample in the discriminator D, and there is also no cross-entropy calculation step. The code is implemented as follows:

```
def g_loss_fn(generator, discriminator, batch_z, is_training):
    # Generator loss function
    fake_image = generator(batch_z, is_training)
    d_fake_logits = discriminator(fake_image, is_training)
    # WGAN-GP G loss function. Maximize the output value of false samples
    loss = - tf.reduce_mean(d_fake_logits)

    return loss
```

Comparing with the original GAN, the main training logic of WGAN is basically the same. The role of the discriminator D for WGAN is a measure of EM distance. Therefore, the more accurate the discriminator is, the more beneficial it is to the generator. The discriminator D can be trained multiple times for a step, and the generator G can trained once to obtain a more accurate EM distance estimation.

13.9 References

[1] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville and Y. Bengio, “Generative Adversarial Nets,” *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence and K. Q. Weinberger, Curran Associates, Inc., 2014, pp. 2672-2680.

[2] A. Radford, L. Metz and S. Chintala, *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks,* 2015.

[3] X. Chen, Y. Duan, R. Houthooft, J. Schulman, I. Sutskever and P. Abbeel, “InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets,” *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon and R. Garnett, Curran Associates, Inc., 2016, pp. 2172-2180.

[4] J.-Y. Zhu, T. Park, P. Isola and A. A. Efros, “Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks,” *Computer Vision (ICCV), 2017 IEEE International Conference on*, 2017.

[5] M. Arjovsky, S. Chintala and L. Bottou, “Wasserstein Generative Adversarial Networks,” 出处 *Proceedings of the 34th International Conference on Machine Learning*, International Convention Centre, Sydney, Australia, 2017.

[6] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin and A. C. Courville, “Improved Training of Wasserstein GANs,” *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan 和 R. Garnett, Curran Associates, Inc., 2017, pp. 5767-5777.

[7] M. Lucic, K. Kurach, M. Michalski, O. Bousquet and S. Gelly, “Are GANs Created Equal? A Large-scale Study,” *Proceedings of the 32Nd International Conference on Neural Information Processing Systems*, USA, 2018.

[8] H. Zhang, I. Goodfellow, D. Metaxas and A. Odena, “Self-Attention Generative Adversarial Networks,” *Proceedings of the 36th International Conference on Machine Learning*, Long Beach, California, USA, 2019.

[9] A. Brock, J. Donahue and K. Simonyan, “Large Scale GAN Training for High Fidelity Natural Image Synthesis,” *International Conference on Learning Representations*, 2019.

[10] L. Metz, B. Poole, D. Pfau and J. Sohl-Dickstein, “Unrolled Generative Adversarial Networks,” *CoRR,* abs/1611.02163, 2016.