Chapter 12

Generative Adversarial Networks

In generative modelling, given a finite set of training samples $\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(N)}\}$ which are drawn from an unknown distribution $p_{data}(\mathbf{x})$, our goal is to learn a parameterized distribution $p_{model}(\mathbf{x})$ that approximates $p_{data}(\mathbf{x})$ as closely as possible. With the learned distribution we can generate samples. For example, a variational auto-encoder, presented in the previous chapter, learns the distribution $p_{model}(\mathbf{x})$ *explicitly* by a neural network.

In this chapter, we will introduce generative adversarial networks (GANs), which learn the distribution $p_{model}(\mathbf{x})$ *implicitly*. An implicit generative model does not directly estimate or fit the distribution. Instead, it produces data instances which approximately follow the underlying distribution.

The GANs are based on a game between two models typically implemented using neural networks. One network called the *generator* defines $p_{model}(\mathbf{x})$ implicitly. The generator is defined by a function $G(z;\theta_g): z \to \mathbf{x}$, where θ_g is a set of learnable parameters and z is an input noise variable on a prior distribution $p_z(z)$ (e.g., uniform or standard normal distribution $\mathcal{N}(\mathbf{0}, \mathbf{I})$). The main role of the generator is to transform such noise z into realistic samples. To learn the generator, another network called the *discriminator* $D(\mathbf{x};\theta_d)$ is required to provide feedback about how *realistic* the samples from the generator are. The discriminator itself also needs to learn its parameters θ_d as a traditional classifier so that it can distinguish the real data from the generated data if the modeled distribution is apart from the underlying distribution.

This chapter covers:

- The principle of the original GAN from a mathematical point of view, including algorithm and convergence
- General framework for GAN training
- Transposed convolutional neural networks, which are important ingredients for image generation.
- A few variants of GANs: conditional GAN, InfoGAN, Wasserstein GAN, CycleGAN
- An example of GAN on MNIST dataset in PyTorch

12.1 Mathematical Description of Original GAN

12.1.1 Principle and Algorithm

The generative adversarial nets (GANs) were originally proposed in Goodfellow et al. (2014). A GAN consists of two basic models: generator $G(z;\theta_g)$ and discriminator $D(\mathbf{x};\theta_d)$, shown in Fig.12.1. Both models can be implemented by neural networks with learnable parameters θ_g and θ_d, respectively. The discriminator $D(\mathbf{x};\theta_d)$ estimates the probability of x coming from the underlying distribution $p_{data}(\mathbf{x})$ rather than the generator-specified distribution $p_g(\mathbf{x})$. It is trained to distinguish the generated samples from the real ones. On the other hand, the generator maps an input noise variable z to the data space, and thus generates a synthetic (or fake) data sample. It is trained to generate fake samples as real as possible so that they can be wrongly recognized by the optimal discriminator as a real sample.

Mathematically, the training process can be described by the following two-player minimax game with an objective function V(D,G):

$$\min_G \max_D \left\{ V(D,G) = \mathrm{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})}[logD(\mathbf{x};\ \theta_d)] + \mathrm{E}_{z\sim p_Z(z)}\left[\log\left(1 - D\big(G\big(z;\ \theta_g\big);\ \theta_d\big)\right)\right]\right\} \quad (12.1)$$

To avoid clutter in the notations, we may drop parameters in the context, e.g. $D(\mathbf{x})$ for $D(\mathbf{x};\ \theta_d)$. The inner max loop is to optimize the discriminator for a given generator while the outer min loop is to learn the parameters θ_g so that the discriminator gives a high output for a fake input. In practice, we usually implement the optimization using an iterative and numerical approach. Instead of fully optimizing in both inner and outer loops, we alternate between *k* steps of maximizing D and one step of minimizing G. This results in D being maintained near its optimal solution, so long as G changes slowly enough.

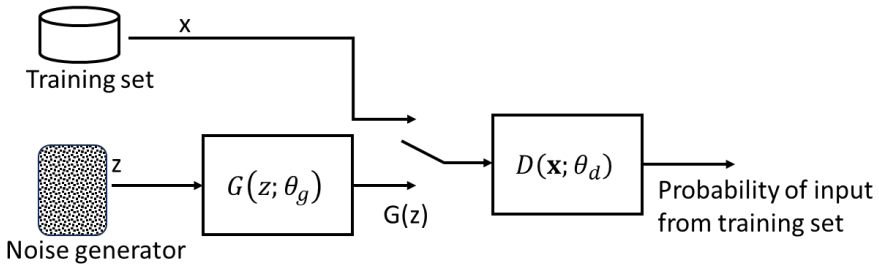


Fig.12.1 The basic architecture of GANs

The algorithm for (12.1) was proposed by Goodfellow et al. (2014) as follows.

Algorithm 1 GAN: Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, *k*, is a hyperparameter. k = 1, the least expensive option, was used in the experiments in Goodfellow et al. (2014).

for number of training iterations **do**

(# part 1: update the discriminator)

for *k* steps **do**

- Sample minibatch of *m* noise samples $\{z^{(1)}, \ldots, z^{(m)}\}$ from noise prior $p_z(z)$.
- Sample minibatch of *m* examples $\{x^{(1)}, \ldots, x^{(m)}\}$ from data generating distribution $p_{data}(x)$.
- Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[logD\big(x^{(i)}\big) + log\left(1 - D\left(G\big(z^{(i)}\big)\right)\right)\right] \quad (12.2)$$

end for

(# part 2: update the generator)

- Sample minibatch of *m* noise samples$\{z^{(1)}, \ldots, z^{(m)}\}$ from noise prior $p_z(z)$.
- Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} log\left(1 - D\left(G\big(z^{(i)}\big)\right)\right) \quad (12.3)$$

end for

12.1.2 Convergence of GANs

In this section, we will investigate the theoretical convergence of the original GANs. We first consider the optimal discriminator for a given generator. Let $p_g(\mathbf{x})$ be the density function defined by the generator, and $p_{data}(\mathbf{x})$ be the density function of the data. We can show that the optimal discriminator on these two distributions is given by (see exercises)

$$D_g^*(\mathbf{x}) = \frac{p_{data}(\mathbf{x})}{p_{data}(\mathbf{x}) + p_g(\mathbf{x})} \tag{12.4}$$

Then, with the optimal discriminator, the minimax game in (12.1) can be rewritten as

$$\min_G \max_D \{V(D,G)\} = \min_G V\left(D_g^*, G\right) \tag{12.5}$$

where

$$\begin{aligned} V\left(D_g^*, G\right) &= \mathrm{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})}\left[log D_g^*(\mathbf{x})\right] + \mathrm{E}_{z\sim p_Z(z)}\left[\log\left(1 - D_g^*\left(G(z)\right)\right)\right] \\ &= \int \left[p_{data}(\mathbf{x}) log D_g^*(\mathbf{x}) + p_g(\mathbf{x}) \log\left(1 - D_g^*(\mathbf{x})\right)\right] d\mathbf{x} \\ &= \int \left[p_{data}(\mathbf{x}) log \frac{p_{data}(\mathbf{x})}{p_{data}(\mathbf{x}) + p_g(\mathbf{x})} + p_g(\mathbf{x}) \log \frac{p_g(\mathbf{x})}{p_{data}(\mathbf{x}) + p_g(\mathbf{x})}\right] d\mathbf{x} \\ &= \mathrm{KL}\left(p_{data}(\mathbf{x}) || \frac{p_{data}(\mathbf{x}) + p_g(\mathbf{x})}{2}\right) + \mathrm{KL}\left(p_g(\mathbf{x}) || \frac{p_{data}(\mathbf{x}) + p_g(\mathbf{x})}{2}\right) - log4 \\ &= 2D_{JS}\left(p_{data}(\mathbf{x}) || p_g(\mathbf{x})\right) - log4 \end{aligned} \tag{12.6}$$

where KL(p||q) denotes the KL divergence between two density function p and q, defined by

$$\mathrm{KL}(p||q) = \int p(x) log \frac{p(x)}{q(x)} dx \tag{12.7}$$

and $D_{JS}(p||q\ )$ is the Jensen-Shannon (JS) divergence, defined by

$$D_{JS}(p||q\ ) = \frac{1}{2}\mathrm{KL}\left(p|| \frac{p+q}{2}\right) + \frac{1}{2}\mathrm{KL}\left(q|| \frac{p+q}{2}\right) \tag{12.8}$$

Since the JS divergence (or KL divergence) is always non-negative and equal to zero if and only if two distributions are identical, the global minimum of $V\left(D_g^*, G\right)$ is $-log4$ which is achieved when $p_g(\mathbf{x}) = p_{data}(\mathbf{x})$, i.e., the generator generates samples perfectly matching the underlying distribution.

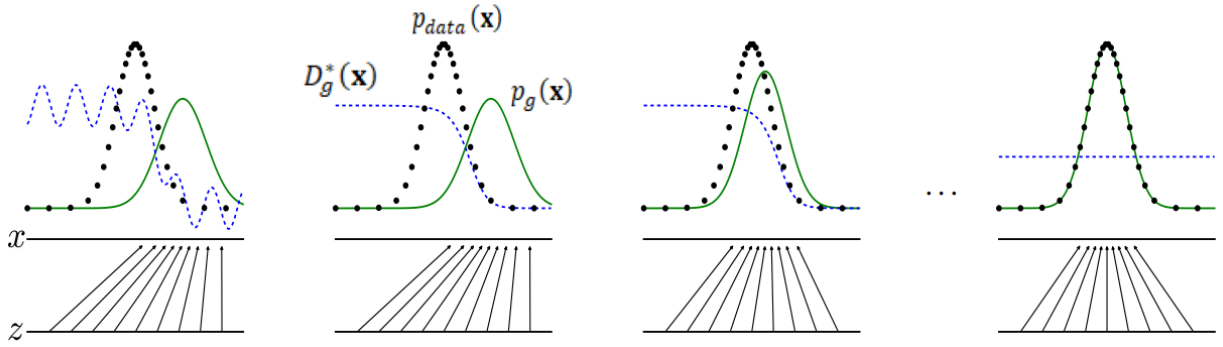
$$V(D^*, G^*) = \min_G \max_D \{V(D,G)\} = \min_G V\left(D_g^*, G\right) = -log4 \tag{12.9}$$

This leads to a conclusion regarding the convergence of the GAN algorithm. If G and D have enough capacity, and at each step of the algorithm the discriminator is allowed to reach its optimum given G, and $p_g(\mathbf{x})$ is updated so as to improve the minimization of

$$\mathrm{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})}\left[log D_g^*(\mathbf{x})\right] + \mathrm{E}_{\mathbf{x}\sim p_g(\mathbf{x})}\left[\log\left(1 - D_g^*(\mathbf{x})\right)\right] \tag{12.10}$$

Then $p_g(\mathbf{x})$ converges to $p_{data}(\mathbf{x})$.

A pedagogical explanation is illustrated in Fig. 4. The GANs are trained by simultaneously updating the discriminative distribution (D, blue dashed line) so that it discriminates between samples from the data generating distribution (black, dotted line) p_x from those of the generative distribution $p_g(G)$ (green, solid line). The lower horizontal line is the domain from which z is sampled, in this case uniformly. The horizontal line above is part of the domain of x. the upward arrows show how the mapping $x = G(z)$ imposes the non-uniform distribution p_g on transformed samples. G contracts in regions of high density and expands in regions of low density of p_g.



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

Fig.12.2 An explanation of GANs [printed from Goodfellow et al. (2014)]. (a) consider an adversarial pair near convergence: p_g is similar to p_{data} and D is a partially accurate classifier. (b) in the inner loop of the algorithm D is trained to discriminate samples from data, converging to $D^*(\mathbf{x}) = \frac{p_{data}(\mathbf{x})}{p_{data}(\mathbf{x})+p_g(\mathbf{x})}$. (c) after an update to G, gradient of D has guided G(z) to flow to regions that are more likely to be classified as data. (d) after several steps of training, if G and D have enough capacity, they will reach a point at which both cannot improve because $p_g = p_{data}$. The discriminator is unable to differentiate between the two distributions, i.e. $D(\mathbf{x}) = 0.5$.

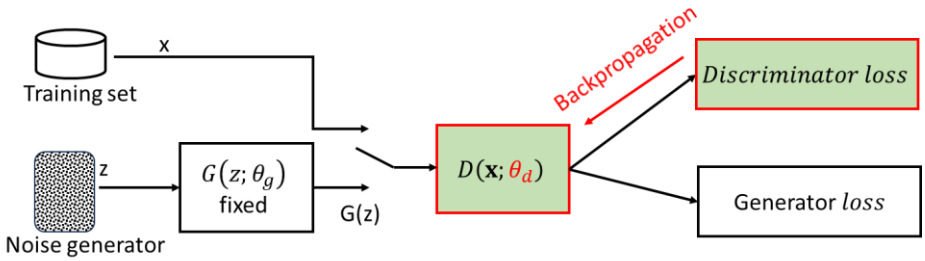
12.2 Implementation of GANs

The previous section presents the basic principle of GANs. In this section, we will describe the details of GAN training and network structure from a practical perspective.

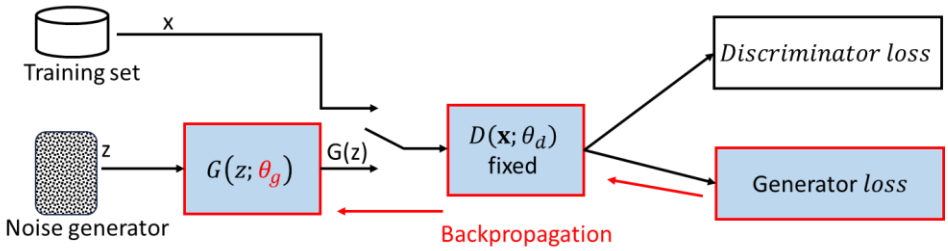
12.2.1 Alternating two training processes

A generative adversarial network (GAN) has two parts: 1) the generator learns to generate fake data samples which are negative training examples for the discriminator; and 2) the discriminator learns to distinguish the generator's fake data from real data (i.e. positive examples). The discriminator penalizes the generator for producing implausible results.

We usually train the generator and the discriminator separately, as shown in Fig.12.3, in an alternative manner: (a) The discriminator trains for one or more batches/epochs. (b) The generator trains for one or more batches/epochs. Repeat steps 1) and 2) to continue to train the generator and discriminator networks. The generator is fixed during the discriminator training phase. Similarly, we keep the discriminator unchanged during the generator training phase.



(a) Discriminator training



(b) Generator training

Fig.12.3 Training of GANs (the shadowed boxes are involved in backpropagation)

A) Discriminator training

The discriminator in a GAN is simply a classifier. It tries to distinguish real data from the data created by the generator. It could use any network architecture appropriate to the type of data it's classifying. The discriminator's training data comes from two sources: 1) Real data instances as positive examples, such as real pictures of people; and 2) Fake data instances created by the generator as negative examples.

The goal of discriminator training is to maximize the objective function (12.1) for a given generator. In practice, we usually update the discriminator parameters using a standard gradient descent optimizer (e.g. Adam in PyTorch) by minimizing a loss function. Obviously, the corresponding loss function is the Binary Cross Entropy loss (BCELoss) function. For the *i*-th input of the discriminator, let the label be $y^{(i)} = 1$ for a real sample and $y^{(i)} = 0$ for a fake data sample. The BCELoss function is defined as

$$Discriminator\ loss = BCEloss = -\sum_{i=1}^{N}\left[y^{(i)} log D^{(i)} + \left(1 - y^{(i)}\right) log\left(1 - D^{(i)}\right)\right] \tag{12.11}$$

where $D^{(i)}$ denotes the prediction (i.e. output) of the discriminator for the *i*-th input (real or fake sample), and *N* is the batch size.

In summary, the discriminator training process can be described as:

- The discriminator predicts $D^{(i)}$ for both real data samples and fake data samples.
- The loss function (12.11) is calculated.
- The parameters of the discriminator are updated by a gradient descent optimizer based on the gradients of the loss function.

B) Generator training

The generator training is shown in Fig.12.3(b). The generator learns to create fake data samples so that the discriminator classifies its output as real for the fake samples. The generator loss penalizes the generator for producing a sample that the discriminator network classifies correctly as fake. According to (12.1), the goal of the generator training is to minimize $\log\left(1 - D\left(G\left(z;\ \theta_g\right);\ \theta_d\right)\right)$ with θ_d fixed. Since this objective function may not provide sufficient gradients, especially in the earlier training stage, we equivalently maximize $\log D\left(G\left(z;\ \theta_g\right)\right)$ which is further equivalent to minimizing $-\log D\left(G\left(z;\ \theta_g\right)\right)$. Thus, the generator loss function can be specified by

$$Generator\ loss = -\sum_{i=1}^{N} \log D\left(G\left(z^{(i)};\ \theta_g\right)\right) \tag{12.12}$$

Note that the real data samples do not participate in the generator training (12.12). To implement (12.12) by BCELoss in PyTorch, we just need to define all the labels for the faked samples as $y^{(i)} = 1$.

During generator training, we keep the discriminator fixed. So we train the generator with the following procedure:

- Sample random noise.
- Generate the fake samples from sampled random noise.
- Compute the prediction on the fake samples.
- Calculate loss (12.12).
- Update the generator parameters with gradient descent optimizer.

12.2.2 Transposed convolutional neural networks

It is straightforward to implement both the generator and the discriminator using neural networks. The discriminator is simply a binary classifier that can be implemented by a series of standard 2D convolutional layers. The generator delivers a fake data sample (e.g., image) for a given input noise vector. Since the size of the noise vector is typically much smaller than that of the data sample (e.g., image), up-sampling layers are required in the generative neural network. In this section, we will present a type of up-sampling convolutional layer, called *transposed convolutional* layer that generates the output feature map greater than the input feature map. For simplicity, in the following presentation we assume single channel in both the input and the filter (or kernel). We can generalize to multiple channels in the same way as we do for standard convolution. In other words, the output for multi-channel input convolution is the sum of convolutions of individual channels.

A) Standard convolution matrix

A standard convolution can be expressed as a multiplication between a convolution matrix and an input vector. Consider an input map feature 4x4 is convoluted with a filter/kernel 3x3 with stride 1 and no zero-padding, shown in Fig.12.4.

The convolution can be represented by

$$Y = \overline{W} \times X \tag{12.13}$$

where $\overline{W} \in \mathbb{R}^{4\times 16}$ is the convolution matrix obtained by re-arranging the weight matrix $W \in \mathbb{R}^{3\times 3}$

$$\overline{W} = \begin{pmatrix} w00 & w01 & w02 & 0 & w10 & w11 & w12 & 0 & w20 & w21 & w22 & 0 & 0 & 0 & 0 & 0 \\ 0 & w00 & w01 & w02 & 0 & w10 & w11 & w12 & 0 & w20 & w21 & w22 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & w00 & w01 & w02 & 0 & w10 & w11 & w12 & 0 & w20 & w21 & w22 & 0 \\ 0 & 0 & 0 & 0 & 0 & w00 & w01 & w02 & 0 & w10 & w11 & w12 & 0 & w20 & w21 & w22 \end{pmatrix} \tag{12.14}$$

X and Y are input and output in a vector form

$$X = \begin{pmatrix} x00 \\ x01 \\ x02 \\ x03 \\ x10 \\ x11 \\ \vdots \\ x32 \\ x33 \end{pmatrix} \qquad Y = \begin{pmatrix} y00 \\ y01 \\ y10 \\ y11 \end{pmatrix}$$

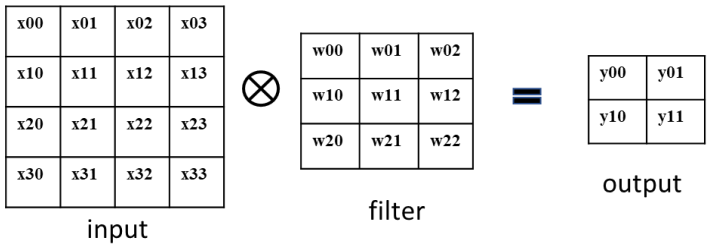


Fig.12.4 A standard convoluion operation (stride 1 and no zero-padding)

B) Transposed convolution

Now let's consider an opposite mapping, i.e., mapping from a 4-dimensional space to a 16-dimensional space. We can define the transposed convolution matrix ($\overline{W}^T \in \mathbb{R}^{16\times4}$ in the above example) by transposing the original convolution matrix $\overline{W}$, such that the transposed convolution matrix can map a 4-dimensional feature to a 16-dimensional feature by a matrix multiplication.

$$Z = \overline{W}^T \times Y \tag{12.15}$$

where transposed convolution matrix $\overline{W}^T \in \mathbb{R}^{16\times4}$, $\overline{W}$ is defined by (12.14), input feature map $Y \in \mathbb{R}^4$, output feature map $Z \in \mathbb{R}^{16}$. The operation in (12.15) is called *transposed convolution* with a filter W, as shown in Fig.12.5, with X and Y reshaped to 2D arrays.

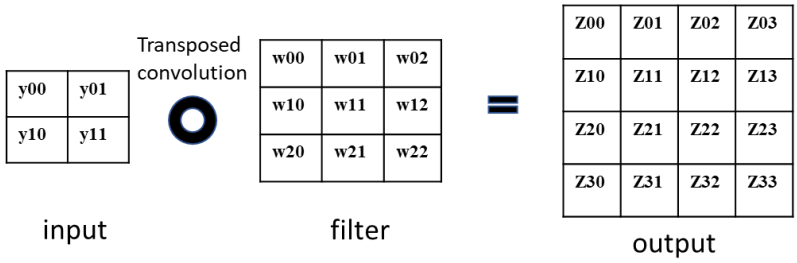


Fig.12.5 A transposed convolution

We can implement the transposed convolution in two perspectives: flipping weight and weighting weight. In the perspective of flipping weight, as shown in Fig.12.6, we flip the weight matrix W verterically and horizontally, and then perform the regular convolution operation between Y (as the input) and the flipped weight matrix (as the kernel), with sufficient zero-padding on Y.

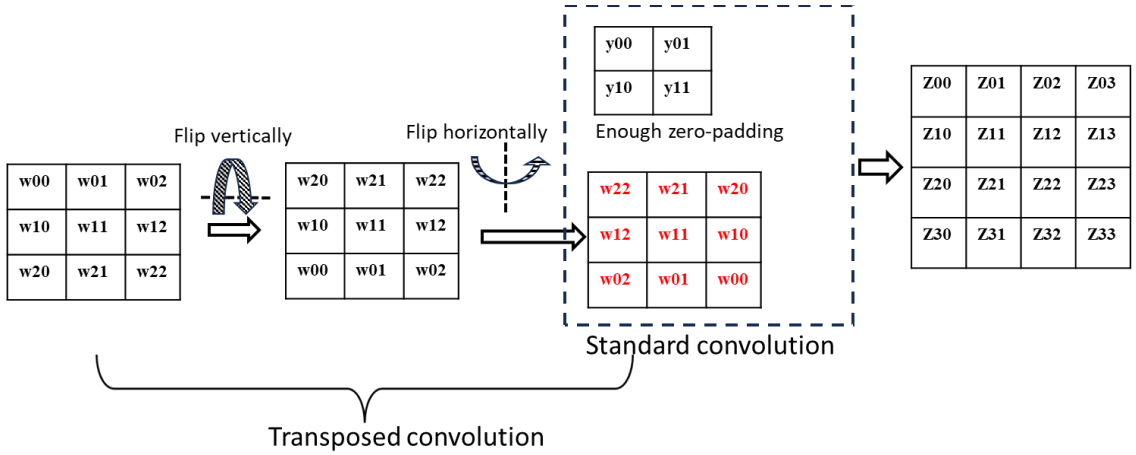


Fig.12.6 Implement a transposed convolution by flipping the filter sheet.

The second perspective for the transposed convolution implementation is shown in Fig.12.7. We generate the partial output maps by sliding the weight matrix multiplied by the corresponding input element, and then sum up all partial output maps in elementwise.

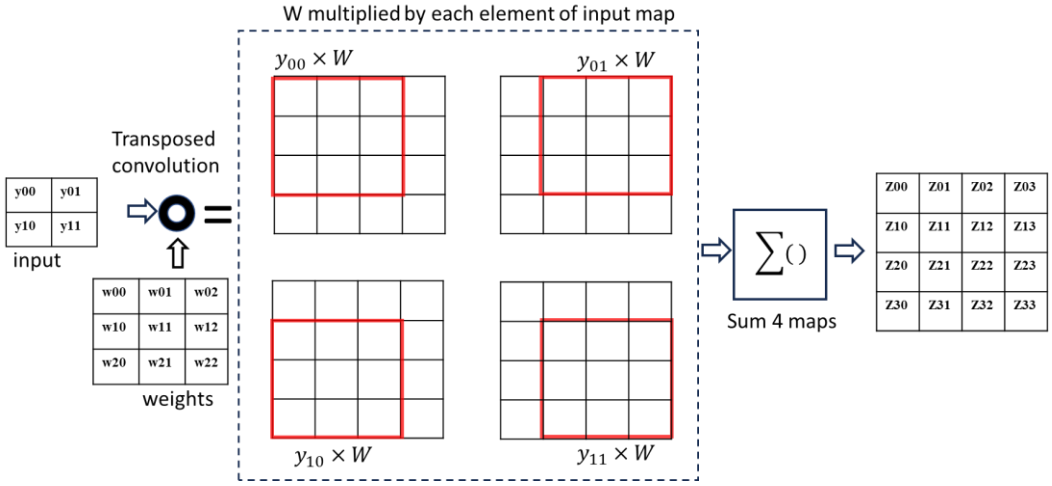


Fig.12.7 Implement a transposed convolution by summing slided weighted filters

Note that, for a default setting, the output feature map size is given by

$$O_{trans\_conv} = n + f - 1 \qquad no\ padding, and\ stride = 1 \qquad (12.16)$$

where *n* is the input size (height or width), *f* is the size of the kernel (height or width).

C) Zero-padding and stride in transposed convolution

The previous description of transposed convolution is based on a default setting, i.e. no zero-padding and stride =1. Like standard convolution, to obtain a desired shape size of output, we can define zero-padding and stride for a transposed convolution.

In a standard convolution, zero-padding simply refers to adding zero around the input feature map while stride means the number of pixels the kernel slide at each step. Zero-padding increases the output size, and increasing the stride leads to down-sampling at the output. Suppose both the shapes of the data feature map and the kernel are square. We have the output shape size ((8.1) in chapter 8)

$$O_{conv} = \left\lfloor \frac{n + 2p - f}{s} \right\rfloor + 1 \tag{12.17}$$

where n is the input size (height or width), p is the number of zero columns/rows padded on one side of the input feature map, f is the size of the kernel, s is the stride.

However, the meanings of "zero-padding" and "stride" specified in a transposed convolution are different. They have the reverse effects on the output size, compared to standard convolution. Similarly, let n be the input size, f be the filter size, p be the zero-padding, and s be the stride for a transposed convolution. In the transposed convolution, zero-padding will decrease the output size by cutting off outer rows and columns of the output. For example, Fig.12.8 shows a transposed convolution with p=1 that gets an output 2×2 by deleting one outer row and column around the default output (non zero-padding). In general, the output size of transposed convolution for zero-padding p (stride =1 default) is given by

$$O_{trans\_conv} = n + f - 1 - 2p \qquad stride = 1 \tag{12.18}$$

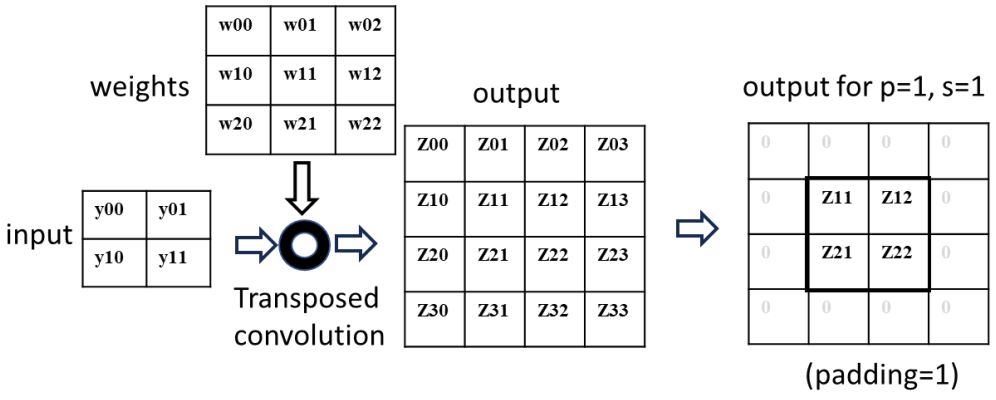


Fig.12.8 Zero-padding p=1 for transposed convolution

The stride specifies the step size by which the weighted weight matrix moves for each input pixel. For example, if stride =2, then the weighted weight matrix moves by two units each step. A stride s (e.g. s=2) is equivalent to inserting $s - 1$ zero(s) between every two input pixels, leading to the effective input size of $n + (n - 1)(s - 1) = s(n - 1) + 1$, shown in Fig.12.9 for s=2.

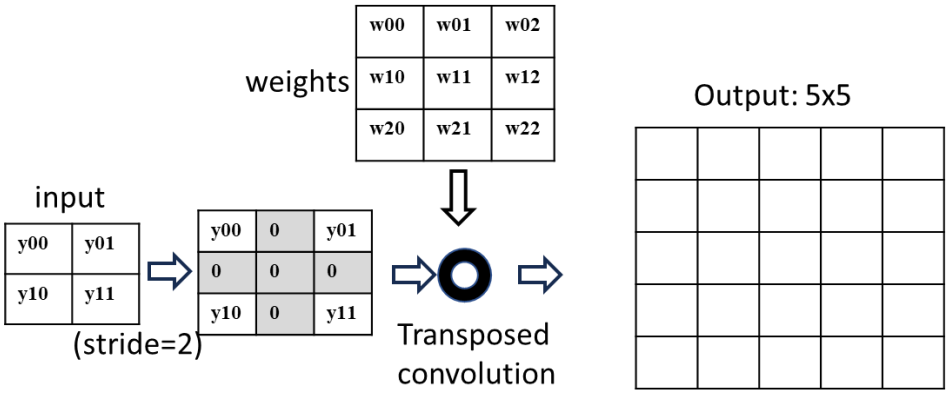


Fig.12.9 A transposed convolution with stride =2

In summary, a transposed convolution with padding p and stride s delivers an output feature map with size given by

$$O_{tranns\_conv} = s(n-1) + f - 2p \tag{12.19}$$

From (12.17) and (12.19), a transposed convolution can reverse the output feature map size of its counterpart standard convolution (i.e., with same kernel size f, padding p and stride s) back to the input size, if $\frac{n+2p-f}{s}$ is an integer, as shown in Fig.12.10.

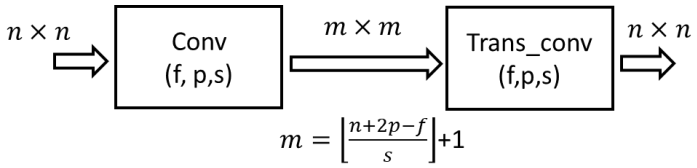


Fig.12.10 Transposed convolution reverses the feature size from standard convolution.

12.2.3 An example of GAN

As an example, in this section we present the detailed neural network architecture for a particular GAN. This GAN is designed to generate 64×64 images. The implementation of this example in PyTorch will be given in Section 12.4.1.

The generator, $G(z;\ \theta_g)$, is designed to map the noise vector $z \in \mathbb{R}^{100}$ to data space $\mathbf{x} \in \mathbb{R}^{3\times64\times64}$, which corresponds to an RGB image. We implement the generator using a series of two-dimensional transposed convolution layers, each (except the last layer) followed by a 2D batch normalization layer and a ReLU activation. The output of the last transposed convolution layer is passed through a tanh function to return the values to the range of $[-1,1]$. The architecture is shown in Fig.12.11.

The discriminator, $D(\mathbf{x};\ \theta_d)$, is a binary classification network that takes an image $\mathbf{x} \in \mathbb{R}^{3\times64\times64}$ as input and outputs a scalar value that indicates the probability that the input image is real (as opposed to fake). Here, the network $D(\mathbf{x};\ \theta_d)$ processes the image through a series of Conv2d, BatchNorm2d, and LeakyReLU layers, and outputs the probability through a Sigmoid activation function. Note that there is no BatchNorm2d for the first convolution layer. The architecture is shown in Fig.12.12.

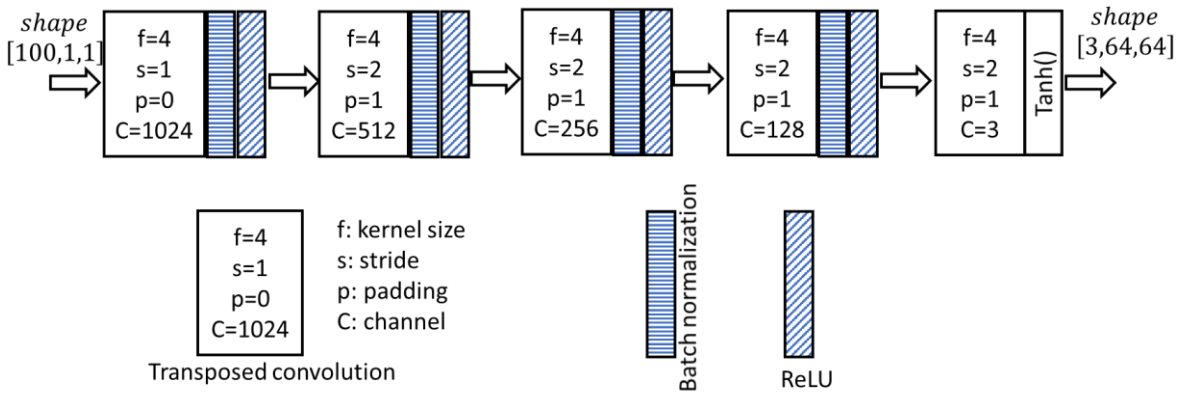


Fig.12.11 Generator implemented by five transposed convolution layers.

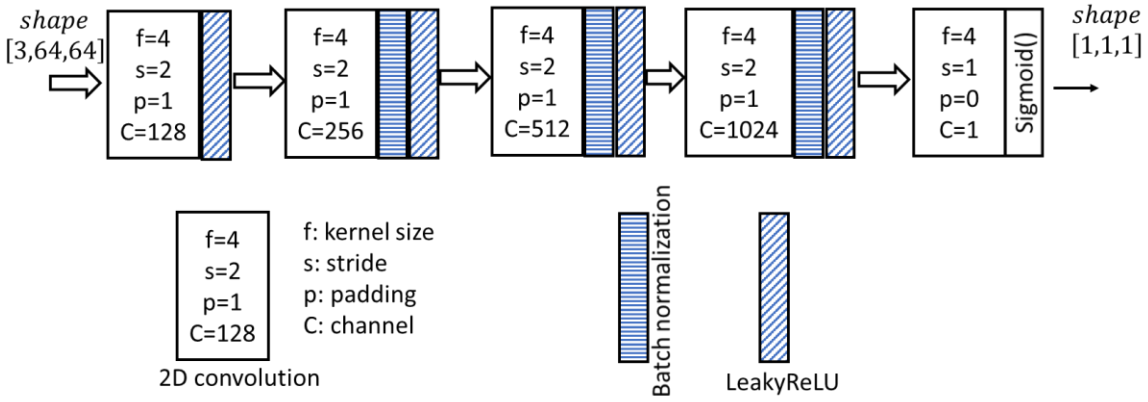


Fig.12.12 Discriminator implemented by five conv2d layers.

12.3 GAN Variants

12.3.1 Practical issues with the original GAN

The probability density learned by the original GAN, $P_g(\mathbf{x})$, is implicitly defined by the neural network $G(z;\ \theta_g)$ that maps a prior random variable $z \sim p(z)$ to a generated sample. In other words, to generate a sample, we first sample z from a simple prior $p(z)$, and then compute $G(z;\ \theta_g)$. During the generation of a sample, we don't have any control over the properties of the sample we want to generate. For instance, we can't ask the original GAN to generate a specific digit (e.g. 7). Conditional GAN and InfoGAN will be introduced to deal with this issue.

In general, GANs suffer the following major problems:

- Non-convergence. The model parameters never converge during the training process.

- Vanishing gradient. When the discriminator is close to its optimum, the generator gradient vanishes (i.e., almost 0) and thus the model learns nothing.
- Model collapse. The trained generator produces limited varieties of samples.
- Highly sensitive to hyperparameter selection.

The in-depth analysis of these problems is beyond the scope of the text. We mention them here to motivate the variants of GAN.

12.3.2 Conditional GAN

(A) Principle and loss function

Generative adversarial nets can be extended to a conditional model if both the generator and discriminator are conditioned on some extra information *c*, which could be any kind of auxiliary information, such as class labels or data from other modalities. We can perform the conditioning by feeding *c* into both the discriminator and the generator as additional input information. This results in a GAN variant called *conditional* GAN. For example, a trained conditional GAN on MNIST can generate an image for a given digit.

In the generator the input noise z, and condition c (e.g., label) are combined. In the discriminator the real image **x** or the fake image G(z|c) and c are presented as the input. Fig.12.13 shows the generic architecture of conditional GANs. The objective function of a two-player minimax game would be as

$$\min_G \max_D \left\{ V(D,G) = \mathrm{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})}[logD(\mathbf{x}|c;\ \theta_d)] + \mathrm{E}_{z\sim p_Z(z)}\left[\log\left(1 - D\big(G\big(z|c;\ \theta_g\big)|c;\ \theta_d\big)\right)\right]\right\} \quad (12.20)$$

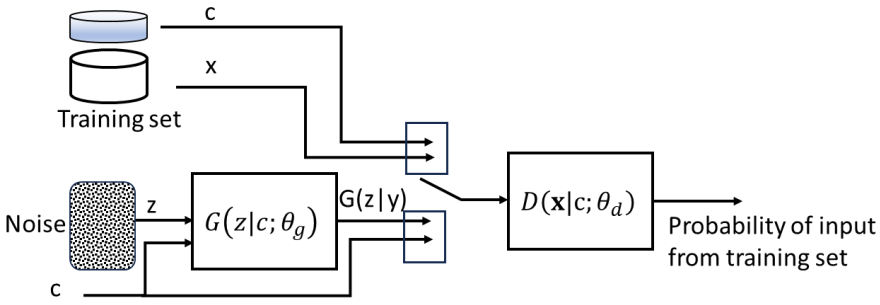


Fig.12.13 Conditional GAN architecture

(B) Implementation

As an example, Fig.12.14 shows the implementation of a conditional GAN for digit image generation, which is conditioned on label $c \in \{0,1,2,3,4,5,6,7,8,9\}$. The input of discriminator (D) consists of an image (real or fake) and its corresponding label image. In our case the size of image x is 32x32. The label image is encoded in one-hot-feature image format 10x32x32. For example, if the label y=3, the third feature map [3,:,:] of the label image is all-one while all other feature maps are all-zero. The generator (G) receives a noise vector z and one-hot label (c) vector as the input, and delivers a fake image for the class c, G(z|c). It is also necessary to encode the label in the one-hot-feature image format to form a fake training example. How the input x or z is combined with the label c is flexible in general. For example, we can choose this way: the first hidden layers for z and c label in the generator are separate, and their outputs are concatenated

(or merged) as the input of the next layer. A similar way is applied to the discriminator. The detailed architecture is given in Table 12.1.

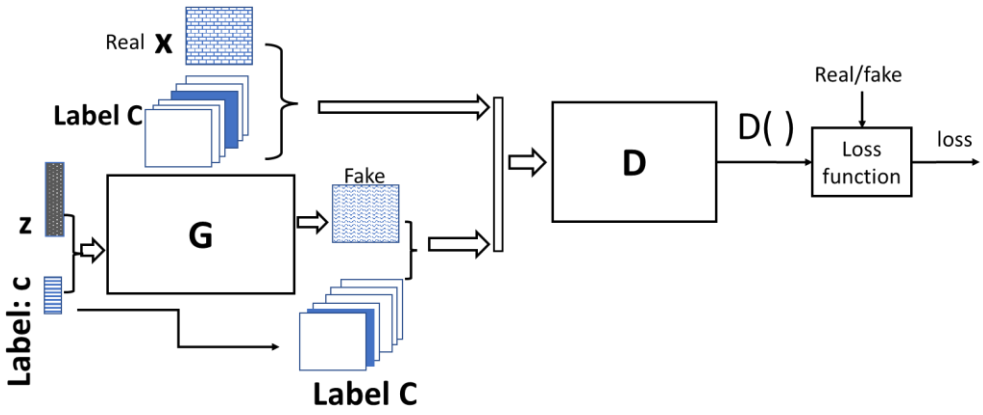


Fig.12.14 Overall architecture of conditional GAN for MNIST

Table 12.1 An example of conditional GAN for MNIST

Generator		**Discriminator**	
Input z [100,1,1]	Input C [10,1,1]	Input x [1,32,32]	Input C [10,32,32]
T.conv(c=256,f=4,s=1,p=0) BN, ReLU	T.conv(c=256,f=4,s=1,p=0) BN, ReLU	Conv(c=64,f=4,s=2,p=1)	Conv(c=64,f=4,s=2,p=1)
Concatenate output: [512,4,4]		Concatenate output: [128,16,16]	
T.conv(c=256,f=4,s=2,p=1) BN, ReLU Output: [256,8,8]		Conv(c=256,f=4,s=2,p=1) BN, LeakyReLU Output: [256,8,8]	
T.conv(c=128,f=4,s=2,p=1) BN, ReLU Output: [128,16,16]		Conv(c=512,f=4,s=2,p=1) BN, LeakyReLU Output: [512,4,4]	
T.conv(c=1,f=4,s=2,p=1) Tanh() Output: [1,32,32]		Conv(c=1,f=4,s=1,p=0) Sigmoid() Output: [1,1,1]	
		Loss for discriminator	Loss for generator

Note: c: channel, f: filter size, s: stride, p: padding, BN: batch normalization.

12.3.3 InfoGAN

(A) Principle and loss function

InfoGAN is similar to conditional GAN except for the fact that it is also able to learn disentangled (interpretable) features in a completely unsupervised manner. Recall, in conditional GAN, the generator network has a conditional input c, which is assumed to be semantically known, e.g., labels. During training, we need to provide c, and G will implicitly learn the conditional distribution of data $p(\mathbf{x}|c)$. In InfoGAN we assume c to be unknown, and thus we infer it based on the data, i.e., we want to find posterior $p(c|\mathbf{x})$.

Instead of using a single noise vector z as in the basic GAN, InfoGAN decomposes noise vector into the following two parts: 1) z, which is treated as incompressible noise as in the basic GAN; 2) c, which is called

the latent vector for the salient structured semantic features of the data distribution. For instance, to generate images from the MNIST dataset, the latent vector $c = [c_1, c_2, c_3]^T$ can be associated with three semantic features: categorical variable $c_1 \sim Categorical(K = 10, p = 0.1)$ for the identity of the digit (i.e., label), continuous variable c_2 for the rotation, and continuous variable c_3 for the thickness of the stroke, $c_2, c_3 \sim Uniform(-1,1)$. In general, the categorical latent variable may be used to control the type or class of the generated image, and the continuous latent variables can capture variations that are continuous in nature.

In information theory, the mutual information between X and Y, measuring the amount of information learned from knowledge of random variable Y about the other random variable X, is defined by

$$I(X;Y) = H(X) - H(X|Y) = H(Y) - H(Y|X) \tag{12.21}$$

In other words, $I(X;Y)$ is the reduction of uncertainty in X when Y is observed. If X and Y are independent, then $I(X;Y)$ is equal to zero. If X and Y are related by a deterministic invertible function, then maximal mutual information is attained. InfoGAN is formulated as an information-regularized minimax game given by

$$\min_G \max_D \{V_I(D,G) = V(D,G) - \lambda I\big(c; G(z,c)\big)\} \tag{12.22}$$

where $I\big(c; G(z,c)\big)$ is the mutual information between c and the generated image, and λ is a hyperparameter.

The intuition of (12.22) is that there should be high mutual information between latent vector c and generator distribution $G(z,c)$. In other words, the information in c should not be lost in the generation process. It has been proven by (Chen, *et al*., 2016) that the mutual information $I\big(c; G(z,c)\big)$ has a lower bound by defining an auxiliary distribution $Q(c|\mathbf{x})$ for an approximation of $p(c|\mathbf{x})$

$$I\big(c; G(z,c)\big) \geq L_I(G,Q) \tag{12.23}$$

where the lower bound

$$L_I(G,Q) = \mathbb{E}_{c\sim p(c), \mathbf{x}\sim G(z,c)}[logQ(c|\mathbf{x})] + H(c) \tag{12.24}$$

In addition, it has been known (Chen, *et al*., 2016) that when the lower bound attains its maximum $L_I(G,Q) = H(c)$ for discrete latent c, the bound becomes tight and the maximal mutual information is achieved.

Thus, (12.22) can be rewritten as

$$\min_{G,Q} \max_D \{V_{InfoGAN}(D,G,Q) = V(D,G) - \lambda L_I(G,Q)\} \tag{12.25}$$

In practice, three components G, D, Q can be implemented as neural networks. This results in the architecture of InfoGAN shown in Fig.12.15. Typically, D and Q share most of convolutional layers while having separate heads to generate corresponding outputs. The D head generates the probability of the input being real, and the Q head computes the conditional distribution $Q(c|\mathbf{x})$.

For a categorical latent component, say c_i, softmax can be used in the output layer for $Q(c_i|\mathbf{x})$. For a continuous latent component, say c_j, $Q\big(c_j|\mathbf{x}\big)$ can be assumed to be Gaussian in many applications, and thus

the neural network only needs to predict the mean and the variance. The hyperparameter λ is typically set to 1 for discrete latent components. For continuous latent components, a smaller λ (e.g., 0.1) is usually used to ensure $\lambda L_I(G,Q)$ is on the same scale as the GAN objectives.

Like the training of the basic GAN, there are two separate parameter updating processes within a training loop for each batch. One process is to update the parameters of discriminator and D head while another process updates the generator and Q head.

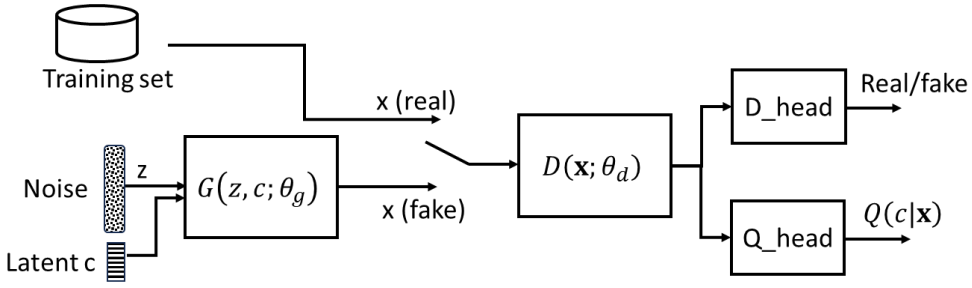


Fig.12.15 Architecture of InfoGAN

(B) Principle and loss function

An example of InfoGAN, which was given by the original authors in (Chen, *et al*., 2016), generates digit images from the MNIST dataset. The architecture is specified in Table 12.1. The latent variables include one categorical variable $c_1 \sim Categorical(K=10, p=0.1)$ and two continuous variables $c_2, c_3 \sim Uniform(-1,1)$ that can capture rotation of digits and stoke width. The noise z is a 62-element random vector. This results in the input size of 74 for the generator. In training, the discriminator and D head are updated using BCE loss from the D head. The loss for updating the generator and Q head is the sum of all three losses listed in Table 12.2. (details: https://github.com/Natsu6767/InfoGAN-PyTorch)

Table 12.2 InfoGAN architecture example for MNIST dataset

<table>
<tr><td>Generator: input [74, 1, 1]</td><td></td><td colspan="3">Discriminator: input [1, 28, 28]</td></tr>
<tr><td>Trans. Conv (c=1024,f=1,s=1,p=0)
BN, ReLU
Output: [1024, 1, 1]</td><td></td><td colspan="3">Conv(c=64,f=4,s=2,p=1)
Leaky ReLU
Output: [64, 14, 14]</td></tr>
<tr><td>Trans. Conv (c=128,f=7,s=1,p=0)
BN, ReLU
Output: [128, 7, 7]</td><td></td><td colspan="3">Conv(c=128,f=4,s=2,p=1)
BN, Leaky ReLU
Output: [128, 7, 7]</td></tr>
<tr><td>Trans. Conv (c=64,f=4,s=2,p=1)
BN, ReLU
Output: [64, 14, 14]</td><td></td><td colspan="3">Conv(c=1024,f=7,s=1,p=0)
BN, Leaky ReLU
Output: [1024, 1, 1]</td></tr>
<tr><td rowspan="3">Trans. Conv(c=1,f=4,s=2,p=1)
Sigmoid
Output: [1, 28, 28]</td><td rowspan="3"></td><td>D head</td><td colspan="2">Q head</td></tr>
<tr><td>Conv(c=1,f=1,s=1,p=0)
Sigmoid
Output: [1,1,1]</td><td colspan="2">Conv(c=128,f=1,s=1,p=0)
BN, Leaky ReLU
Output: [128,1,1]</td></tr>
<tr><td></td><td>Conv1</td><td>Conv2</td><td>Conv3
exp()</td></tr>
<tr><td></td><td></td><td>Binary cross entropy (BCE) loss</td><td>Cross entropy loss</td><td colspan="2">Gaussian negative log likelihood loss</td></tr>
</table>

Note: c: channel, f: filter size, s: stride, p: padding, BN: batch normalization.
Conv1: c=10, f=1,s=1,p=0, output size: [10]
Conv2: c=2, f=1,s=1,p=0, output size: [2] for Gaussian means
Conv3: c=2, f=1,s=1,p=0, output size: [2] for Gaussian variances

12.3.4 Wasserstein GAN

Wasserstein GAN (WGAN) (Arjovsky *et al.*, 2017) was proposed to treat the instability of GAN training by an objective function based on Wasserstein distance, which has better theoretical properties than the original one (12.1).

(A) Wasserstein distance and WGAN principle

To explain the concept of Wasserstein distance, let's consider a 2-dimensional example of moving all dirt in piles (indicated by cuboids) to some pre-defined empty holes (indicated by cylinders), as illustrated in Fig.12.16. The amount of dirt across the piles represents the source distribution $p_r(x, y)$ (the subscript r indicates it is associated with real data in GAN) while the resulting dirt distribution across the holes represents the target distribution $p_\theta(x, y)$.

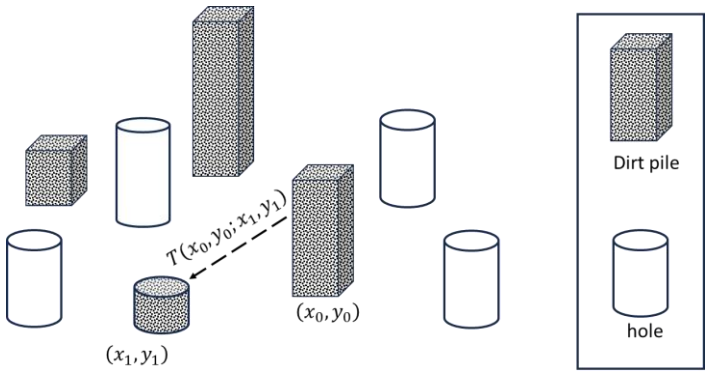


Fig.12.16 Wasserstein distance illustration by dirt transportation

Suppose both distributions $p_r(x, y)$ and $p_\theta(x, y)$ are given. The goal is to find the most efficient transportation plan, i.e., the optimal one, which minimizes the total transportation cost. The cost should be proportional to the amount dirt transported and the moving distance. To quantify the cost, we define the squared Euclidean distance as the cost of moving one unit of dirt from (x_1, x_0) to (y_1, y_0)

$$C(x_0, y_0; x_1, y_1) = (x_1 - x_0)^2 + (y_1 - y_0)^2 \tag{12.26}$$

Then we define a transportation plan $T(x_0, y_0; x_1, y_1)$ for all possible (x_0, y_0) and (x_1, y_1), which specifies how many units of dirt to move from (x_0, y_0) to (x_1, y_1). A pile of dirt is allowed to split into multiple holes, and the dirt from multiple piles can be moved into the same hole. For T to be a valid plan, it should be non-negative, i.e. $T(x_0, y_0; x_1, y_1) \geq 0$ and

$$\iint T(x_0, y_0; x_1, y_1) dx_1 dy_1 = p_r(x_0, y_0) \qquad for\ all\ (x_0, y_0) \tag{12.27a}$$

$$\iint T(x_0, y_0; x_1, y_1) dx_0 dy_0 = p_\theta(x_1, y_1) \qquad for\ all\ (x_1, y_1) \tag{12.27b}$$

(12.27a) implies that all dirt at (x_0, y_0) are transported to some holes while (12.27b) tells that all dirt in a hole (x_1, y_1) came from piles. Thus, for a given transportation plan $T(x_0, y_0; x_1, y_1)$, the total cost of moving dirt from piles $p_r(x, y)$ to holes $p_\theta(x, y)$ is given by

$$Cost = \iint \iint C(x_0, y_0; x_1, y_1) T(x_0, y_0; x_1, y_1) dx_0 dy_0 \, dx_1 dy_1 \tag{12.28}$$

The Wasserstein distance is the minimal cost corresponding to the optimal transportation plan.

In general, the Wasserstein distance, also called Earth-Mover (EM) distance, between two distributions $p_r(\mathbf{x})$ and $p_\theta(\mathbf{y})$, is defined by

$$W(p_r, p_\theta) = \inf_{\mathcal{T} \in \prod(p_r, p_\theta)} \mathbb{E}_{(\mathbf{x},\mathbf{y}) \sim \mathcal{T}}[\|\mathbf{x} - \mathbf{y}\|] \tag{12.29}$$

where $\mathcal{T} \in \prod(p_r, p_\theta)$ denotes the set of all joint distributions $T(\mathbf{x}; \mathbf{y})$ whose marginal distributions are p_r and p_θ respectively. inf denotes the infimum operation (i.e., the greatest lower bound). Intuitively $T(\mathbf{x}; \mathbf{y})$ indicates how much probability mass needs to be transported from **x** to **y** in order to transform the distribution p_r to p_θ. Thus, to obtain p_θ from p_r, the Wasserstein distance is the total mass required to move by the optimal transport plan corresponding to the optimal $T(\mathbf{x}; \mathbf{y})$.

Wasserstein distance demonstrates a property of *continuity*, which results in smooth and non-vanishing gradients in the learning process. Suppose we model a distribution by parameter θ, denoted as p_θ. A sequence of distributions $p_{\theta_t}, t = 1,2, ...,$ converges if and only if there is a distribution p_∞ such that $distance(p_{\theta_t}, p_\infty)$ tends to zero. Continuity means that when a sequence of parameter θ_t converge to θ, the distributions p_{θ_t} also converge to p_θ. Note that the continuity depends on the definition of the distance. For example, consider two probability mass functions $p_0(x)$ and $p_\theta(x)$ in Fig.12.17, where the random variable X is discrete, and $p_\theta(x)$ is the shifted version of $p_0(x)$ by θ. One can verify that the Wasserstein distance between them is θ. Thus, the distribution $p_\theta(x)$ converges to $p_0(x)$ when $\theta \to 0$, in the sense of Wasserstein distance, which is expected by our intuition. However, their KL or JS divergence does not have this continuity property, because their values are given as

$$D_{JS}(p_0, p_\theta) = \begin{cases} log2 & \theta \neq 0 \\ 0 & \theta = 0 \end{cases} \tag{12.30}$$

$$D_{KL}(p_0, p_\theta) = D_{KL}(p_\theta, p_0) = \begin{cases} +\infty & \theta \neq 0 \\ 0 & \theta = 0 \end{cases} \tag{12.31}$$

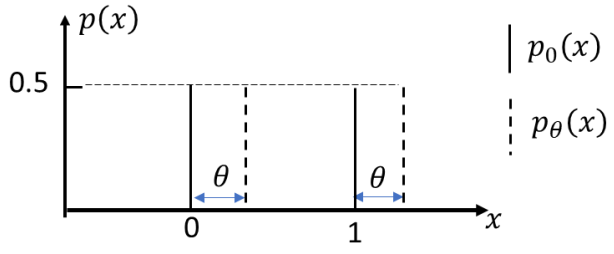


Fig.12.17 two distributions for divergence comparison

In Wasserstein GAN, the Wasserstein distance is used to form the loss for training. The goal of training is to minimize the Wasserstein distance, instead of minimizing the JS divergence like the original GAN (12.6).

However, the Wasserstein distance is highly intractable. Using the Kantorovich-Rubinstein duality (Villani 2009), we can simplify (12.29) to

$$W(p_r, p_\theta) = \sup_{\|f\|_L \leq 1} \mathbb{E}_{x \sim p_r}[f(x)] - \mathbb{E}_{x \sim p_\theta}[f(x)] \tag{12.32}$$

where the supremum, i.e., the least upper bound, denoted by $\sup_{\|f\|_L \leq 1}$, is over all the 1-Lipschitz functions $f: X \to \mathbb{R}$. A real function $f: X \to \mathbb{R}$ is a K-Lipschitz function if

$$|f(x_1) - f(x_2)| \leq K \cdot |x_1 - x_2|, \ for\ any\ x_1, x_2 \in X \tag{12.33}$$

Intuitively, the constraint of Lipschitz limits the changing rate (or derivative) of the function.

Under the constraint of Lipschitz, WGAN can be described by

$$\min_G \max_{D \in \mathcal{D}} \mathbb{E}_{x \sim p_r}[D(x; w)] - \mathbb{E}_{\tilde{x} \sim p_\theta}[D(\tilde{x}; w)] \tag{12.34}$$

where $\mathcal{D}$ is the set of 1-Lipschitz functions, and p_θ is the model distribution implicitly defined by $\tilde{x} = G(z; \theta), z \sim p(z)$. Thus, (12.34) can be rewritten as

$$\min_G \max_{D \in \mathcal{D}} \mathbb{E}_{x \sim p_r}[D(x; w)] - \mathbb{E}_{z \sim p(z)}[D(G(z; \theta); w)] \tag{12.35}$$

(B) WGAN with weight clipping

The discriminator $D(x; w)$ in WGAN is called a *critic* since it is not trained to classify. Under an optimal discriminator, minimizing (12.35) with respect to G minimizes the Wasserstein distance $W(p_r, p_\theta)$. With mild assumptions, it has been shown (Arjovsky *et al.*, 2017) that

$$\nabla_\theta W(p_r, p_\theta) = -\mathbb{E}_{z \sim p(z)}[\nabla_\theta D(G(z; \theta); w)] \tag{12.36}$$

To enforce the Lipschitz constraint on the critic, the original authors (Arjovsky *et al.*, 2017) proposed to clip the weights of the critic to lie within a compact space [-c,c], say c=0.01. the set of function satisfying this constraint is a subset of the Lipschitz functions which depends on c and the critic architecture. The resulting WGAN procedure is described in Algorithm 2. An example of WGAN architecture is specified in Table 12.3, for generating [3,64.64] images.

Algorithm 2: WGAN with weight clipping. Suggested hyperparameters: $\alpha = 0.00005$, $c = 0.01$, $m = 64$, $n_{critic} = 5$. Suggested optimizer: RMSProp.

Require: α, the learning rate. c, the clipping parameter, m, the batch size. n_{critic}, the number of iterations of the critic per generator iteration.

Require: w_0, initial critic parameters. θ_0, initial generator parameters.

Procedure:

while θ has not converged **do**
  for t = 1, …, n_{critic} **do**

Sample $\{x^{(i)}\}_{i=1}^{m} \sim p_r$ a batch from the real data.

Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.

$g_w \leftarrow \nabla_w \left[\frac{1}{m}\sum_{i=1}^{m} D\left(x^{(i)}; w\right) - \frac{1}{m}\sum_{i=1}^{m} D\left(G\left(z^{(i)}; \theta\right); w\right)\right]$

$w \leftarrow w + \alpha \cdot RMSProp(w, g_w)$

$w \leftarrow clip(w, -c, c)$

end for

Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.

$g_\theta \leftarrow -\nabla_\theta \frac{1}{m}\sum_{i=1}^{m} D\left(G\left(z^{(i)}; \theta\right); w\right)$

$\theta \leftarrow \theta - \alpha \cdot RMSProp(\theta, g_\theta)$

end while

Table 12.3 WGAN-weight clipping for generating $3 \times d \times d$ images (d=64 for celebA dataset)

Generator	Critic
Input z [100,1,1]	Input image [3,64,64]
T.conv2d(cin=100,cout=8×d,f=4,s=2,p=0) BN, ReLU	Conv2d(cin=3,cout=d,f=4,s=2,p=1) BN, LeakyReLU
T.conv2d(cin=8×d,cout=4×d,f=4,s=2,p=1) BN, ReLU	Conv2d(cin=d,cout=2×d,f=4,s=2,p=1) BN, LeakyReLU
T.conv2d(cin=4×d,cout=2×d,f=4,s=2,p=1) BN, ReLU	Conv2d(cin=2×d,cout=4×d,f=4,s=2,p=1) BN, LeakyReLU
T.conv2d(cin=2×d,cout=d,f=4,s=2,p=1) BN, ReLU	Conv2d(cin=4×d,cout=8×d,f=4,s=2,p=1) BN, LeakyReLU
T.conv2d(cin=d,cout=3,f=4,s=2,p=1) Tanh()	Conv2d(cin=8×d,cout=1,f=4,s=1,p=0)

(C) WGAN with gradient penalty

Enforcing the Lipschitz constraint via weight clipping biases the critic towards much simpler functions. The trained critic ignores higher moments of the data distribution and thus models very simple approximation to the optimal functions. The convergence of WGAN with weight clipping is sensitive to the value of c.

Gulrajani *et al*., 2017 proposed an alternative way to enforce the Lipschitz constraint using gradient penalty. Since a differentiable function is 1-Lipschitz if and only if it has gradients with norm at most 1 everywhere, we consider constraining the gradient norm of the critic's output with respect to its input. To avoid tractability issues, a soft version of the constraint with a penalty on the gradient norm for random samples $\hat{x} \sim p_{\hat{x}}$. The loss function for the critic in WGAN with gradient penalty is

$$L = \mathbb{E}_{z\sim p(z)}[D(G(z;\theta);w)] - \mathbb{E}_{x\sim p_r}[D(x;w)] + \lambda \mathbb{E}_{\hat{x}\sim p_{\hat{x}}}[(\|\nabla_{\hat{x}} D(\hat{x})\|_2 - 1)^2] \qquad (12.37)$$

The first two terms in (12.37) are the original critic loss, and the third term is the penalty of gradient.

Sampling $\hat{x} \sim p_{\hat{x}}$ is implicitly defined as sampling uniformly along line segments between pairs of points sampled from the real data distribution p_r and the generator distribution p_g. As a result, the WGAN with gradient penalty is described in Algorithm 3.

Since we penalize the norm of the critic's gradient with respect to each input independently, it is suggested that the batch normalization is not applied in the critic model. Instead, an instance normalization can be considered.

Algorithm 3: WGAN with gradient penalty. Suggested hyperparameters: $\lambda = 10$, $\alpha = 0.0001$, $\beta_1 = 0.5$, $\beta_2 = 0.999$, $n_{critic} = 5$. Suggested optimizer: Adam.
Require: the gradient penalty coefficient λ, the batch size m, the number of iterations of the critic per generator iteration n_{critic}, Adam hyperparameters α, β_1, β_2.
Require: initial critic parameters w_0, initial generator parameters θ_0.
Procedure:

while θ has not converged **do**
  for t = 1, …, n_{critic} **do**
    for i =1,…, m **do**
      Sample real data $x \sim p_r$, prior $z \sim p(z)$, a random number $\epsilon \sim U[0,1]$.
      $\tilde{x} \leftarrow G(z;\theta)$
      $\hat{x} \leftarrow \epsilon x + (1-\epsilon)\tilde{x}$
      $L^{(i)} \leftarrow D(\tilde{x};w) - D(x;w) + \lambda(\|\nabla_{\hat{x}} D(\hat{x};w)\|_2 - 1)^2$
    end for
    $w \leftarrow \text{Adam}\left(\nabla_w \frac{1}{m}\sum_{i=1}^{m} L^{(i)}, w, \alpha, \beta_1, \beta_2\right)$
  end for
  Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
  $\theta \leftarrow \text{Adam}\left(\nabla_\theta \frac{1}{m}\sum_{i=1}^{m} -D\left(G\left(z^{(i)};\theta\right);w\right), \theta, \alpha, \beta_1, \beta_2\right)$
end while

As an example, to implement the WGAN with gradient penalty, we can still use Table 12.3 and just need to delete or replace BN layers in the critic with instance normalization layers (nn.InstanceNorm2d). Detailed similar implementations (including Python codes and training) can be found in many githubs (e.g., Zeleni9, 2021, and s-chh, 2022).

12.3.5 CycleGAN

(A) Principle and loss function

CycleGAN (Zhu *et al*. 2017) was proposed to perform image-to-image translations. For example, a CycleGAN can translate a photo to an image in Monet painting style while maintaining the content. Other

image-to-image translation tasks include changing the season from summer to winter, translating horse into zebra, changing dark to night, etc. The goal of CycleGAN is to learn the special characteristics of one image collection and figure out how these characteristics could be translated into the other image collection.

Suppose we have two image collections (or domains) X and Y: $\{x^{(i)}\}_{i=1}^{N}, x^{(i)} \in X$ and $\{y^{(j)}\}_{j=1}^{M}, y^{(j)} \in Y$. We denote the data distributions as $x \sim p_{data}(x)$ and $y \sim p_{data}(y)$. The goal is to learn two mapping functions $G: X \to Y$, and $F: Y \to X$, such that $G(x) \sim p_{data}(y)$ and $F(y) \sim p_{data}(x)$ approximately, and $F\big(G(x)\big) \approx x$ and $G\big(F(y)\big) \approx y$. The mapping F can be interpreted as the inverse mapping of G.

Thus, the CycleGAN model consists of two generators G and F, and their corresponding discriminators D_Y and D_X, as shown in Fig.12.18. The discriminator D_Y aims to distinguish between $y^{(j)}$ as real and $G(x^{(i)})$ as fake while the discriminator D_X in the same way aims to predict $x^{(i)}$ as real and $F(y^{(j)})$ as fake.

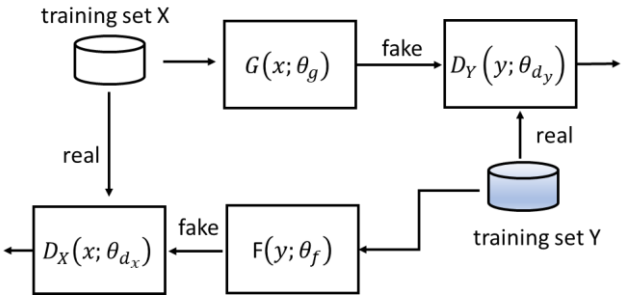


Fig.12.18 Model diagram of CycleGAN

The objective function contains two types of losses: adversarial losses and cycle consistency losses. For the mapping function G and its discriminator D_Y, the objective can be expressed as

$$\mathcal{L}_{GAN}(G, D_Y, X, Y) = \mathbb{E}_{y \sim p_{data}(y)}[\log D_Y(y)] + \mathbb{E}_{x \sim p_{data}(x)}\left[\log\left(1 - D_Y\big(G(x)\big)\right)\right] \tag{12.38}$$

G aims to minimize the objective against the discriminator that tries to maximize it, i.e.,

$$\min_G \max_{D_Y} \mathcal{L}_{GAN}(G, D_Y, X, Y) \tag{12.39}$$

Similarly, for the mapping function F and its discriminator D_X, we have an objective

$$\mathcal{L}_{GAN}(F, D_X, Y, X) = \mathbb{E}_{x \sim p_{data}(x)}[\log D_X(x)] + \mathbb{E}_{y \sim p_{data}(y)}\left[\log\left(1 - D_X\big(F(y)\big)\right)\right] \tag{12.40}$$

for the minmax optimization,

$$\min_F \max_{D_X} \mathcal{L}_{GAN}(F, D_X, Y, X) \tag{12.41}$$

Theoretically, the adversarial losses alone cannot guarantee that the learned function can map an individual input $x^{(i)}$ to a desired output $y^{(i)}$. A cycle consistency loss is constructed to enforce the *cycle consistency*. The cycle consistency means that the learned mapping function should be cycle-consistent: for each image $x \in X$, the image translation cycle should bring *x* back to the original image, i.e., $x \to G(x) \to F\big(G(x)\big) \approx x$, and $y \to F(y) \to G\big(F(y)\big) \approx y$. The consistency loss is defined as

$$\mathcal{L}_{cyc}(G, F) = \mathbb{E}_{x \sim p_{data}(x)}\left[\left\|F\big(G(x)\big) - x\right\|_1\right] + \mathbb{E}_{y \sim p_{data}(y)}\left[\left\|G\big(F(y)\big) - y\right\|_1\right] \tag{12.42}$$

Optionally, to preserve the color style of the input image, an identity loss is defined as

$$\mathcal{L}_{identity}(G,F) = \mathbb{E}_{x\sim p_{data}(x)}[\|F(x)-x\|_1] + \mathbb{E}_{y\sim p_{data}(y)}[\|G(y)-y\|_1] \tag{12.43}$$

The identity loss pushes the generator to be near an identity mapping when real samples of the target domain are provided as the input to the generator.

By combining (12.38) (12.40) (12.42) and (12.43), we obtain the full objective for CycleGAN

$$\mathcal{L}(G,F,D_X,D_Y) = \mathcal{L}_{GAN}(G,D_Y,X,Y) + \mathcal{L}_{GAN}(F,D_X,Y,X) + \lambda_1\mathcal{L}_{cyc}(G,F) + \lambda_2\mathcal{L}_{identity}(G,F) \tag{12.44}$$

where the hyperparameters λ_1, λ_2 control the relative importance of the two types of losses (e.g., $\lambda_1 = 10, \lambda_2 = 5$). The training process is to solve the optimization problem,

$$\min_{G,F}\max_{D_X,D_Y} \mathcal{L}(G,F,D_X,D_Y) \tag{12.45}$$

To stabilize the training procedure, the original authors suggested that the negative log likelihood objective in (12.38) and (12.40) be replaced by a least-squares loss.

Based on the above discussion, the loss functions for training CycleGAN are described explicitly below:

1) Discriminator D_Y loss

$$\mathcal{L}_{D_Y} = \mathbb{E}_{y\sim p_{data}(y)}\left[\left(1-D_Y(y)\right)^2\right] + \mathbb{E}_{x\sim p_{data}(x)}\left[\left(D_Y\left(G(x)\right)\right)^2\right] \tag{12.46a}$$

2) Discriminator D_X loss

$$\mathcal{L}_{D_X} = \mathbb{E}_{x\sim p_{data}(x)}\left[\left(1-D_X(x)\right)^2\right] + \mathbb{E}_{y\sim p_{data}(y)}\left[\left(D_X\left(F(y)\right)\right)^2\right] \tag{12.46b}$$

3) Generator G loss:

$$\mathcal{L}_G = \mathbb{E}_{x\sim p_{data}(x)}\left[\left(1-D_Y\left(G(x)\right)\right)^2\right] + \lambda_1\mathcal{L}_{cyc}(G,F) + \lambda_2\mathcal{L}_{identity}(G,F) \tag{12.46c}$$

4) Generator F loss

$$\mathcal{L}_F = \mathbb{E}_{y\sim p_{data}(y)}\left[\left(1-D_X\left(F(y)\right)\right)^2\right] + \lambda_1\mathcal{L}_{cyc}(G,F) + \lambda_2\mathcal{L}_{identity}(G,F) \tag{12.46d}$$

(B) Implementation

The network architectures for the generators and the discriminators, suggested by Zhu *et al*. 2017, is detailed in Table 12.4. The input image size is assumed to be [3, k, k] with k=128 or 256 or higher.

In the generator, the first three convolutional layers function as an *encoder*, which learns a representation with an increased number of channels. The resulting activation is then passed to a series of 6 or 9 residual blocks (6 for k=128, and 9 for k=256 or higher), called *transformer*. It is then expanded by a *decoder*, which uses two transposed convolutional layers to enlarge the representation size with the reduced number of channels, and one convolutional output layer to produce the final image.

The discriminator, originally proposed by Isola et al., 2017, delivers an array output, instead of a single scalar. Each element in the output array corresponds to a receptive field or a patch with a size N×N in the input image. This discriminator tries to classify if each N×N patch in an image is real or fake. The average of elements in the output provides the ultimate output of the discriminator, i.e., the image is real or fake. For the setting in Table 12.4, the patch size is 70×70. Thus, the architecture of the discriminator is called 70×70 *PatchGAN*. Given an image of size 256x256, the discriminator outputs a tensor of size 30x30. Each value of the output tensor holds the classification result (real or fake) for a 70x70 area of the input image. Note that these 70x70 areas overlap with each other. This is equivalent to manually selecting each of these 70x70 areas and having the discriminator examine them iteratively. The final classification result on the whole image is the average of classification results on the 30x30 values.

Table 12.4 Network architectures for CycleGAN (from Zhu *et al*. 2017)

Generator (G and F)		**Discriminator (D_Y and D_X)**
Input: [3,k,k]		Input image [3,k,k]
Conv2d(cin=3, cout=64,f=7,s=1,p=3) InstanceNorm2d, ReLU (output: [64,k,k]		Conv2d(cin=3,cout=64,f=4,s=2,p=1) LeakyReLU (output: [64, k/2, k/2])
Conv2d(cin=64, cout=128,f=3,s=2,p=1) InstanceNorm2d, ReLU (output: [128,k/2,k/2])		Conv2d(cin=64,cout=128,f=4,s=2,p=1) InstanceNorm2d, LeakyReLU (output: [128,k/4,k/4])
Conv2d(cin=128, cout=256,f=3,s=2,p=1) InstanceNorm2d, ReLU (output: [256,k/4,k/4])		Conv2d(cin=128,cout=256,f=4,s=2,p=1) InstanceNorm2d, LeakyReLU (output:[256,k/8,k/8])
6 or 9 Residual blocks*	Conv2d(cin=256,cout=256,f=3,s=1,p=1) InstanceNorm2d, ReLU Conv2d(cin=256,cout=256,f=3,s=1,p=1) InstanceNorm2d Block input + InstanceNorm2d output (output: [256,k/4,k/4])	Conv2d(cin=256,cout=512,f=4,s=1,p=1) InstanceNorm2d, LeakyReLU (output:[512,k/8-1, k/8-1])
ConvTranspose2d(cin=256, cout=128,f=3,s=2,p=1) InstanceNorm2d, ReLU (output: [128, k/2,k/2])		Conv2d(cin=512,cout=1,f=4,s=1,p=1) Sigmoid() (output: [1, k/8-2, k/8-2])
ConvTranspose2d(cin=128, cout=64,f=3,s=2,p=1) InstanceNorm2d, ReLU (output: [64,k,k]		
Conv2d(cin=64, cout=3,f=7,s=1,p=3) Tanh() (output:[3,k,k])		

*6 residual blocks for 128x128 input images (k=128) while 9 residual blocks for k=256.

12.3.6 *f*-GANs

f-GANs are a class of GANs defined by various divergences. In fact, the original GAN is a special case of *f*-GANs.

(A) *f*-divergences

f-divergences (Liese *et al*. 2006), also known as the Ali-Silvey distances (Ali *et al*.1966), are a class of divergences that measure the difference between two probability distributions. We shall see that KL and

Jensen-Shannon divergences are examples of *f*-divergences. Given two distributions P and Q that possess, respectively, an absolutely continuous density function $p(x)$ and $q(x)$ on the domain $\mathcal{X}$, the *f*-divergence is defined as

$$D_f(p||q) = \mathbb{E}_{x\sim q}\left[f\left(\frac{p(x)}{q(x)}\right)\right] \tag{12.47}$$

where the *f*-divergence function $f: \mathbb{R}_+ \to \mathbb{R}$ is a convex and lower semi-continuous function satisfying $f(1) = 0$. Different choices result in different divergences. The convex (or called Fenchel) conjugate of f is defined as

$$f^*(t) = \sup_{u\in dom_f}\{ut - f(u)\} \tag{12.48}$$

The function f^* is also a convex and lower semi-continuous function. The pair (f, f^*) is dual in sense that $f^{**} = f$.

(B) Variational divergence minimization

The lower bound on the *f*-divergence is obtained as below.

$$\begin{aligned} D_f(p||q) &= \int_{\mathcal{X}} q(x) \sup_{t\in dom_{f^*}}\left\{t\frac{p(x)}{q(x)} - f^*(t)\right\}dx \\ &\geq \sup_{T\in\mathcal{T}}\left\{\int_{\mathcal{X}} p(x)T(x)dx - \int_{\mathcal{X}} q(x)f^*\big(T(x)\big)dx\right\} \\ &= \sup_{T\in\mathcal{T}}\{\mathbb{E}_{x\sim p}[T(x)] - \mathbb{E}_{x\sim q}\big[f^*\big(T(x)\big)\big]\} \end{aligned} \tag{12.49}$$

where $\mathcal{T}$ is an arbitrary class of functions $T: \mathcal{X} \to \mathbb{R}$. The inequality in (12.49) is introduced for two reasons: 1) Jensen's inequality when swapping the integration and supremum operations; and 2) $\mathcal{T}$ may contain only a subset of all possible functions. T is also called a variational function.

Under mild conditions on *f* (Nguyen *et al*, 2010), the bound in (12.49) is tight for

$$T(x) = \frac{df(u)}{du}|_{u=\frac{p(x)}{q(x)}} \tag{12.50}$$

Now we use two neural networks T_ω and Q_θ to model the variational function and distribution Q, respectively. According to the lower bound in (12.49), our goal is to learn the generative model Q_θ by finding a saddle-point of the following *f*-GAN objective function, which we minimize w.r.t θ and maximize w.r.t ω,

$$F(\theta, \omega) = \mathbb{E}_{x\sim p}[T_\omega(x)] - \mathbb{E}_{x\sim Q_\theta}\big[f^*\big(T_\omega(x)\big)\big] \tag{12.51}$$

To match dom_{f^*}, the domain of the conjugate function f^*, the variational function $T_\omega(x)$ is further represented in the form $T_\omega(x) = g_f\big(V_\omega(x)\big)$, and thus the objective function (12.51) is rewritten as

$$F(\theta, \omega) = \mathbb{E}_{x\sim p}\big[g_f\big(V_\omega(x)\big)\big] + \mathbb{E}_{x\sim Q_\theta}\left[-f^*\left(g_f\big(V_\omega(x)\big)\right)\right] \tag{12.52}$$

where $V_\omega(x): \mathcal{X} \to \mathbb{R}$ and $g_f: \mathbb{R} \to dom_{f^*}$ is an output activation function specific to the *f*-divergence. It is suggested that g_f is chosen to be a monotonically increasing function. We estimate $\mathbb{E}_{x\sim p}[g_f(V_\omega(x))]$ by sampling a mini-batch from a training set, and estimate $\mathbb{E}_{x\sim Q_\theta}\left[-f^*\left(g_f(V_\omega(x))\right)\right]$ by sampling from the generative model Q_θ. Table 12.5 lists some *f*-divergences and the relevant functions.

Table 12.5 Some *f*-divergences, and the corresponding functions $f(u)$ and conjugates $f^*(t)$, and recommended output activation functions $g_f(v)$. (Duplicated from Nowozin *et al*, 2016)

Name	$\boldsymbol{D_f(p\|\|q)}$	$\boldsymbol{f(u)}$	$\boldsymbol{f^*(t)}$	$\boldsymbol{dom_{f^*}}$	$\boldsymbol{g_f(v)}$
Total variation	$\frac{1}{2}\int \lvert p(x)-q(x)\rvert\, dx$	$\frac{1}{2}\lvert u-1\rvert$	t	$\left[-\frac{1}{2},\frac{1}{2}\right]$	$\frac{1}{2}\tanh(v)$
KL $D_{KL}(p\|\|q)$	$\int p(x)\log\frac{p(x)}{q(x)}dx$	$u\log u$	e^{t-1}	$\mathbb{R}$	v
Reverse KL	$\int q(x)\log\frac{q(x)}{p(x)}dx$	$-\log u$	$-1-\log(-t)$	$\mathbb{R}_-$	$-e^{-v}$
Pearson $\mathcal{X}^2$	$\int \frac{(q(x)-p(x))^2}{p(x)}dx$	$(u-1)^2$	$\frac{1}{4}t^2+t$	$\mathbb{R}$	v
Neyman $\mathcal{X}^2$	$\int \frac{(p(x)-q(x))^2}{q(x)}dx$	$\frac{(u-1)^2}{u}$	$2-2\sqrt{1-t}$	$t<1$	$1-e^{-v}$
Hallinger	$\int \left(\sqrt{p(x)}-\sqrt{q(x)}\right)^2 dx$	$\left(\sqrt{u}-1\right)^2$	$\frac{t}{1-t}$	$t<1$	$1-e^{-v}$
Jensen-Shannon $D_{JS}(p\|\|q)$	$\frac{1}{2}\left(D_{KL}\left(p\|\|\frac{p+q}{2}\right)+D_{KL}\left(q\|\|\frac{p+q}{2}\right)\right)$	$-(u+1)\log\frac{u+1}{2}+u\log u$	$-\log(2-e^t)$	$t<\log 2$	$\log 2-\log(1+e^{-v})$
Basic GAN	$2D_{JS}(p\|\|q)-\log 4$	$u\log u-(u+1)\log(u+1)$	$-\log(1-e^t)$	$\mathbb{R}_-$	$-\log(1+e^{-v})$

Note: for basic GAN, $f(1)=-\log 4\neq 0$

(C) Basic GAN: a special case of f-divergence model

Given the functions $f(u), f^*(t)$ and $g_f(v)$ for the basic GAN in the last row at Table 12.5, we can have the objective function (12.52) as

$$\begin{aligned} F(\theta,\omega) &= \mathbb{E}_{x\sim p}\left[-\log\left(1+e^{-V_\omega(x)}\right)\right]+\mathbb{E}_{x\sim Q_\theta}\left[-f^*\left(-\log\left(1+e^{-V_\omega(x)}\right)\right)\right] \\ &= \mathbb{E}_{x\sim p}\left[\log\left(\frac{1}{1+e^{-V_\omega(x)}}\right)\right]+\mathbb{E}_{x\sim Q_\theta}\left[-f^*\left(\log\left(\frac{1}{1+e^{-V_\omega(x)}}\right)\right)\right] \\ &= \mathbb{E}_{x\sim p}\left[\log\left(\sigma\left(V_\omega(x)\right)\right)\right]+\mathbb{E}_{x\sim Q_\theta}\left[\log\left(1-\sigma\left(V_\omega(x)\right)\right)\right] \\ &= \mathbb{E}_{x\sim p}\left[\log\left(D_\omega(x)\right)\right]+\mathbb{E}_{x\sim Q_\theta}\left[\log\left(1-D_\omega(x)\right)\right] \end{aligned} \tag{12.53}$$

where $D_\omega(x)=\sigma\left(V_\omega(x)\right)$ is the discriminator in the GAN with a sigmoid activation function. Note that (12.53) is the same as (12.1).

(D) Algorithm of *f*-GANs

A gradient method algorithm for *f*-GANs is suggested by Nowozin *et al*, 2016, which is described below.

Algorithm: single-step gradient method for *f*-GANs

function SingleStepGradientIteration $(P, \theta^t, \omega^t, B, \eta)$,

where P is the training set, B is the batch size, η is the learning rate.

Sample $X_P = \{x_1, x_{2,\ldots,} x_B\}$ and $X_Q = \{x'_1, x'_{2,\ldots,} x'_B\}$ from P and Q_{θ^t}, respectively.

Update: $\omega^{t+1} = \omega^t + \eta \nabla_\omega F(\theta^t, \omega^t)$

Update: $\theta^{t+1} = \theta^t - \eta \nabla_\theta F(\theta^t, \omega^t)$

end function

To speed up the training process as we do in the basic GAN, instead of minimizing $\mathbb{E}_{x\sim Q_\theta}\left[-f^*\left(g_f(V_\omega(x))\right)\right]$ w.r.t θ, we maximize $\mathbb{E}_{x\sim Q_\theta}[g_f(V_\omega(x))]$. Thus, the update for θ can be modified as

$$\theta^{t+1} = \theta^t + \eta \nabla_\theta \mathbb{E}_{x\sim Q_{\theta^t}}\left[g_f\left(V_{\omega^t}(x)\right)\right] \tag{12.54}$$

The analysis of convergence and the detailed neural network implementations can be found in Nowozin *et al*, 2016.

12.4 Example: Deep Convolutional GAN on MNIST Dataset

In this section, we will present an example and demonstrate how to construct and train a basic deep convolutional GAN (DCGAN). The architecture of DCGAN is based on the work by Radford *et. al*., 2016. The trained GAN can generate images, which resemble the images in the training dataset. The Python programs in this section are run on GPU in Google Colab.

12.4.1 Basic DCGAN

First, we explore the implementation of the basic DCGAN. Specifically, the architectures of the generator and the discriminator are shown in Fig.12.11 and Fig.12.12.

Import packages.

```
from __future__ import print_function
import os
import random
import time
import torch
import torch.nn as nn
import torch.optim as optim
import torch.utils.data
import torchvision.datasets as datasets
import torchvision.transforms as transforms
import torchvision.utils as vutils
import numpy as np
import matplotlib.pyplot as plt
```

```
import torchvision
import torchvision.transforms as transforms

# Set random seed for reproducibility
randomSeed = 10
random.seed(randomSeed)
```

Mount Google drive to colab.

```
from google.colab import drive
drive.mount('/content/drive')
```

Define some hyperparameters.

```
dataroot = "./data"     # root directory for dataset
workers = 2             # number of workers for dataloader
batch_size = 128        # batch size
image_size = 64         # image size
nc = 1                  # image channels, 3 for color
nz = 100                # size of z noise vector
ngf = 64                # size of feature maps in generator
ndf = 64                # size of feature maps in discriminator
lr = 0.0002             # learning rate
beta1 = 0.5             # beta1 hyperparameter for Adam optimizers
ngpu = 1                # number of GPUs available. Use 0 for CPU mode.

num_epochs = 10         # number of training epochs
```

Prepare training dataloader and device.

```
# Create the dataset
transform=transforms.Compose([
                               transforms.Resize(image_size),
                               transforms.ToTensor(),
                               transforms.Normalize((0.5,), (0.5,))])
dataset = torchvision.datasets.MNIST(root=dataroot, train=True,
                                     download=True, transform=transform)
```

```
# Create the dataloader
dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
                                         shuffle=True, num_workers=workers)

# Decide device to run on
device = torch.device("cuda:0" if (torch.cuda.is_available() and ngpu > 0) el
se "cpu")
```

Define Generator

```
# Generator

class Generator(nn.Module):
    def __init__(self, ngpu):
        super(Generator, self).__init__()
```

```
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is Z, going into a convolution
            nn.ConvTranspose2d( nz, ngf * 16, 4, 1, 0, bias=True),
            nn.BatchNorm2d(ngf * 16),
            nn.ReLU(True),
            # output size. (ngf*16=1024) x 4 x 4
            nn.ConvTranspose2d(ngf * 16, ngf * 8, 4, 2, 1, bias=True),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            # output size. (ngf*8=512) x 8 x 8
            nn.ConvTranspose2d( ngf * 8, ngf * 4, 4, 2, 1, bias=True),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            # output size. (ngf*4=256) x 16 x 16
            nn.ConvTranspose2d( ngf * 4, ngf * 2, 4, 2, 1, bias=True),
            nn.BatchNorm2d(ngf*2),
            nn.ReLU(True),
            # output size. (ngf*2=128) x 32 x 32
            nn.ConvTranspose2d( ngf * 2, nc, 4, 2, 1, bias=True),
            nn.Tanh()
            # output size. (nc) x 64 x 64
        )

    def forward(self, input):
        return self.main(input)
```

Define Discriminator

```
# Discriminator
class Discriminator(nn.Module):
    def __init__(self, ngpu):
        super(Discriminator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is (nc) x 64 x 64
            nn.Conv2d(nc, ndf * 2, 4, 2, 1, bias=True),
            nn.LeakyReLU(0.2, inplace=True),
            # output size. (ndf*2=128) x 32 x 32
            nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=True),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),
            # output size. (ndf*4=256) x 16 x 16
            nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=True),
            nn.BatchNorm2d(ndf * 8),
            nn.LeakyReLU(0.2, inplace=True),
            # output size. (ndf*8=512) x 8 x 8
            nn.Conv2d(ndf * 8, ndf * 16, 4, 2, 1, bias=True),
            nn.BatchNorm2d(ndf * 16),
            nn.LeakyReLU(0.2, inplace=True),
            # output size. (ndf*16=1024) x 4 x 4
            nn.Conv2d(ndf * 16, 1, 4, 1, 0, bias=True),
            nn.Sigmoid()
        )
```

```
    def forward(self, input):
        return self.main(input)
```

Weight initialization.
It is crucial to initialize GAN properly for the convergence.

```
# weights initialization called by Gnet and Dnet
def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        nn.init.normal_(m.weight.data, 1.0, 0.02) # or (…,0.0, 0.02)
        nn.init.constant_(m.bias.data, 0)
```

Instantiate generator and discriminator (Gnet and Dnet).

```
# Create the generator
Gnet = Generator(ngpu).to(device)
Gnet.apply(weights_init)
```

```
# Create the Discriminator
Dnet = Discriminator(ngpu).to(device)
Dnet.apply(weights_init)
```

Specify loss function, fixed_noise batch (64 values for z), real label and fake label, optimizers.

```
# Initialize BCELoss function
criterion = nn.BCELoss()

# Establish convention for real and fake labels during training
real_label = 1.
fake_label = 0.

# Setup Adam optimizers for both G and D
optimizerD = optim.Adam(Dnet.parameters(), lr=lr, betas=(beta1, 0.999))
optimizerG = optim.Adam(Gnet.parameters(), lr=lr, betas=(beta1, 0.999))
```

Training process
Each iteration consists of two steps: updating D network and then updating G network. D network is updated based on both real and fake samples while G network is updated through the output of D network based on fake samples only. Please note that the convergence of the training process is not guaranteed on each program run.

```
# Training Loop

# Lists to monitor training progress
#
G_losses = []
D_losses = []
num_epochs=10
print("Starting Training Loop...")

for epoch in range(num_epochs):
```

```
    # For each epoch
    for i, data in enumerate(dataloader, 0):
        # for each batch
        #-------------------------
        # (1) Update D network: maximize log(D(x))+log(1-D(G(z)))
        #-----------------------------
        ## (A) Train with all-real batch
        Dnet.zero_grad()
        # Format batch
        real_imgs = data[0].to(device)
        # data[0]: images, data[1]: labels
        # real_imgs: [b_s, 1, image_size, image_size]
        b_s = real_imgs.size(0)
        label = torch.full((b_s,),real_label,dtype=torch.float,device=device)
        # Calculate output of D
        output = Dnet(real_imgs).view(-1)
        # Calculate loss on the real batch
        errD_real = criterion(output, label)
        # Calculate gradients for D
        errD_real.backward()
        D_x = output.mean().item()

        ## (B)Train with all-fake batch
        # Generate batch of latent vectors
        noise = torch.randn(b_s, nz, 1, 1, device=device)
        # Generate fake image batch by G
        fake = Gnet(noise)
        label.fill_(fake_label)
        # Calculate output of D
        output = Dnet(fake.detach()).view(-1)
        # Calculate D's loss on the all-fake batch
        errD_fake = criterion(output, label)
        # Calculate the gradients on this batch
        errD_fake.backward()
        D_G_z1 = output.mean().item()
        # Add the gradients from the all-real and all-fake batches
        errD = errD_real + errD_fake
        # Update D
        optimizerD.step()

        # ---------------------------------------
        # (2) Update G network: maximize log(D(G(z)))
        # ---------------------------------------
        Gnet.zero_grad()
        label.fill_(real_label)  # fake labels are real for generator cost
        # because G wants to fool D, so we treat the fake as real.
        output = Dnet(fake).view(-1)
        # Calculate G's loss
        errG = criterion(output, label)
        # Calculate gradients for G
        errG.backward()
        D_G_z2 = output.mean().item()
        # Update G
        optimizerG.step()

        # Output training stats
        if i % 50 == 0:
```

```
            print('[%d/%d][%d/%d]\tLoss_D: %.4f\tLoss_G: %.4f\tD(x): %.4f\tD(
                  G(z)): %.4f / %.4f'
                  % (epoch, num_epochs, i, len(dataloader),
                     errD.item(), errG.item(), D_x, D_G_z1, D_G_z2))


        # Save Losses for plotting later
        G_losses.append(errG.item())
        D_losses.append(errD.item())
```

```
Starting Training Loop...
[0/10][0/469]   Loss_D: 1.8566  Loss_G: 8.1812  D(x): 0.3774  D(G(z)): 0.3721 / 0.0005
[0/10][50/469]  Loss_D: 3.6225  Loss_G: 44.1498 D(x): 0.4508  D(G(z)): 0.0000 / 0.0000
[0/10][100/469] Loss_D: 0.7124  Loss_G: 2.9447  D(x): 0.6583  D(G(z)): 0.0011 / 0.1247
[0/10][150/469] Loss_D: 0.8989  Loss_G: 0.9111  D(x): 0.5747  D(G(z)): 0.0461 / 0.5524
[0/10][200/469] Loss_D: 1.4422  Loss_G: 0.8815  D(x): 0.3802  D(G(z)): 0.0102 / 0.5227
[0/10][250/469] Loss_D: 0.8741  Loss_G: 1.0921  D(x): 0.5318  D(G(z)): 0.0283 / 0.4374
...
```

Plot the losses for Gnet and Dnet after training (10 epochs).

```
plt.figure(figsize=(10,5))
plt.title("Generator and Discriminator Loss During Training", fontsize=20)
plt.plot(G_losses,label="G")
plt.plot(D_losses,label="D")
plt.xlabel("iterations", fontsize=20)
plt.ylabel("Loss", fontsize=20)
plt.legend()
plt.show()
```

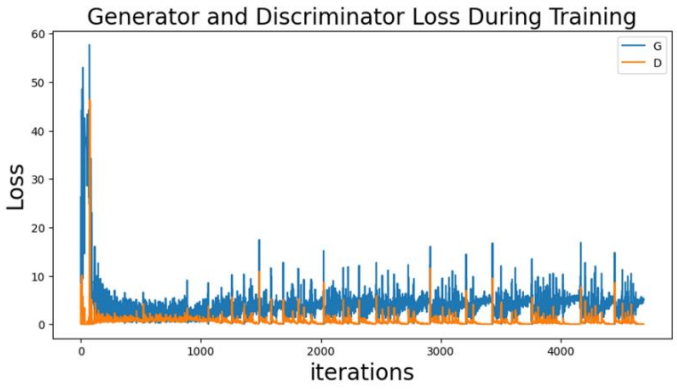


Fig.12.19 Loss plots for D and G for 10 epochs (469 batches per epoch)

Plot real samples and fake samples.

```
fixed_noise = torch.randn(64, nz, 1, 1, device=device)

with torch.no_grad():
  fake = Gnet(fixed_noise).detach().cpu()
img = vutils.make_grid(1-fake, padding=5, normalize=True)

real_batch = next(iter(dataloader))
```

```
# Plot the real images
plt.figure(figsize=(15,15))
plt.subplot(1,2,1)
plt.axis("off")
plt.title("Real Images")
plt.imshow(np.transpose(vutils.make_grid(1-real_batch[0].to(device)[:64], pad
ding=5, normalize=True).cpu(),(1,2,0)))

# Plot the fake images from the last epoch
plt.subplot(1,2,2)
plt.axis("off")
plt.title("Fake Images")
plt.imshow(np.transpose(img,(1,2,0)))
plt.show()
```

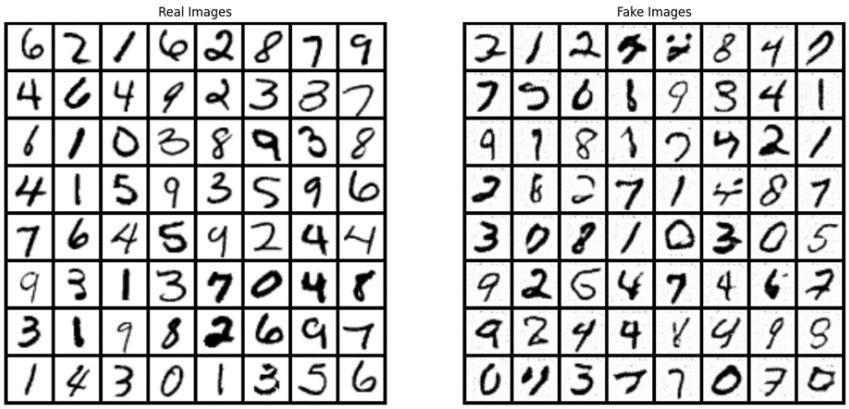


Fig.12.20 Real images versus generated fake images after 10-epoch training

12.4.2 Conditional DCGAN for MNIST dataset

The architecture of the conditional DCGAN implemented in this section is specified in Table 12.1. The development framework is similar to the one for the basic DCGAN in Section 12.4.1. One should pay attention to the differences between them. The major differences lie in: 1) image size, 2) neural network architecture, and 3) an additional input (i.e., condition input) for the generator and the discriminator.

Import packages.

```
from __future__ import print_function
import os
import random
import time
import torch
import torch.nn as nn
import torch.optim as optim
import torch.utils.data
import torchvision.datasets as datasets
import torchvision.transforms as transforms
import torchvision.utils as vutils
import numpy as np
import matplotlib.pyplot as plt
import torchvision
import torchvision.transforms as transforms
```

```
import torch.nn.functional as F

# Set random seed for reproducibility
randomSeed = 10
random.seed(randomSeed)
```

Mount google drive.

```
from google.colab import drive
drive.mount('/content/drive')
```

Specify some hyperparameters.

```
dataroot = "./data"   # directory for dataset
workers = 2           # number of workers for dataloader
batch_size = 128      # batch size
image_size = 32       # image size
nc = 1                # image channels, 3 for color
nz = 100              # size of z noise vector
ngf = 128             # number of feature maps in generator
ndf = 128             # number of feature maps in discriminator
lr = 0.0002           # learning rate
beta1 = 0.5           # beta1 hyperparameter for Adam optimizers
ngpu = 1              # number of GPUs available. Use 0 for CPU mode.

num_epochs = 20       # number of training epochs
```

Prepare dataloader for training process and specify device as GPU or CPU.

```
# Create the dataset
transform=transforms.Compose([
                               transforms.Resize(image_size),
                               transforms.ToTensor(),
                               transforms.Normalize((0.5,), (0.5,))])
dataset = torchvision.datasets.MNIST(root=dataroot, train=True,
                                     download=True, transform=transform)

# Create the dataloader
dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
                                         shuffle=True, num_workers=workers)

# Decide device to run on
device = torch.device("cuda:0" if (torch.cuda.is_available() and ngpu > 0) el
se "cpu")
```

Define the architecture of Generator and Discriminator.

```
# Generator Code

class Generator(nn.Module):
    def __init__(self, ngpu):
        super(Generator, self).__init__()
        self.ngpu = ngpu
        self.deconv1_1 = nn.ConvTranspose2d(100,ngf*2, 4, 1, 0)
        self.deconv1_1_bn = nn.BatchNorm2d(ngf*2)
        self.deconv1_2 = nn.ConvTranspose2d(10, ngf*2, 4, 1, 0)
        self.deconv1_2_bn = nn.BatchNorm2d(ngf*2)
        self.deconv2 = nn.ConvTranspose2d(ngf*4, ngf*2, 4, 2, 1)
```

```
        self.deconv2_bn = nn.BatchNorm2d(ngf*2)
        self.deconv3 = nn.ConvTranspose2d(ngf*2, ngf, 4, 2, 1)
        self.deconv3_bn = nn.BatchNorm2d(ngf)
        self.deconv4 = nn.ConvTranspose2d(ngf, 1, 4, 2, 1)

    def forward(self, input, label):
        x = F.relu(self.deconv1_1_bn(self.deconv1_1(input)))
        y = F.relu(self.deconv1_2_bn(self.deconv1_2(label)))
        x = torch.cat([x,y],1)
        x = F.relu(self.deconv2_bn(self.deconv2(x)))
        x = F.relu(self.deconv3_bn(self.deconv3(x)))
        x = torch.tanh(self.deconv4(x))

        return x
```

```
class Discriminator(nn.Module):
    def __init__(self, ngpu):
        super(Discriminator, self).__init__()
        self.ngpu = ngpu
        self.conv1_1 = nn.Conv2d(1,int(ndf/2), 4, 2, 1)
        self.conv1_2 = nn.Conv2d(10, int(ndf/2), 4, 2, 1)
        self.conv2 = nn.Conv2d(ndf, ndf*2, 4, 2, 1)
        self.conv2_bn = nn.BatchNorm2d(ndf*2)
        self.conv3 = nn.Conv2d(ndf*2, ndf*4, 4, 2, 1)
        self.conv3_bn = nn.BatchNorm2d(ndf*4)
        self.conv4 = nn.Conv2d(ndf*4, 1, 4, 1, 0)


    def forward(self, input, label):
        x = F.leaky_relu(self.conv1_1(input), 0.2)
        y = F.leaky_relu(self.conv1_2(label), 0.2)
        x = torch.cat([x,y], 1)
        x = F.leaky_relu(self.conv2_bn(self.conv2(x)), 0.2)
        x = F.leaky_relu(self.conv3_bn(self.conv3(x)), 0.2)
        x = torch.sigmoid(self.conv4(x))
        return x
```

Initialize the weights.

```
# weights initialization called by Gnet and Dnet
def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        nn.init.normal_(m.weight.data, 1.0, 0.02) # or(..., 0.0,0.02)
        nn.init.constant_(m.bias.data, 0)
```

Instantiate generator as Gnet and discriminator as Dnet, with weight initialization.

```
# Create the generator
Gnet = Generator(ngpu).to(device)
Gnet.apply(weights_init)

# Create the Discriminator
Dnet = Discriminator(ngpu).to(device)
Dnet.apply(weights_init)
```

Specify the loss function and optimizers.

```
# Instantiate BCELoss function
criterion = nn.BCELoss()

# Establish convention for real and fake labels during training
real_label = 1.
fake_label = 0.

# Setup Adam optimizers for both G and D
optimizerD = optim.Adam(Dnet.parameters(), lr=lr, betas=(beta1, 0.999))
optimizerG = optim.Adam(Gnet.parameters(), lr=lr, betas=(beta1, 0.999))
```

Prepare all possible labels in one-hot vector format and in 32x32 one-hot feature map format. The tensor *onehot* will be used to create a one-hot vector for Gnet to generate a conditional image. The tensor fill is used to create a condition image as a part of the input of Dnet.

```
# label preprocess
onehot = torch.zeros(10, 10)
onehot = onehot.scatter_(1, torch.LongTensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]).
view(10,1), 1).view(10, 10, 1, 1)
fill = torch.zeros([10, 10, image_size, image_size])
for i in range(10):
    fill[i, i, :, :] = 1
#onehot [10,10,1,1]:  10 one-hot vectors, each for one digit
# fill [10,10,32,32]: 10 condition images,
#                     each image has 10 feature maps with one-hot feature map.
```

Training loop. The diagram in Fig.12.21 illustrates the tensors used in the training loop and how they are related to the neural networks Gnet and Dnet. It can help a reader understand the training loop code.

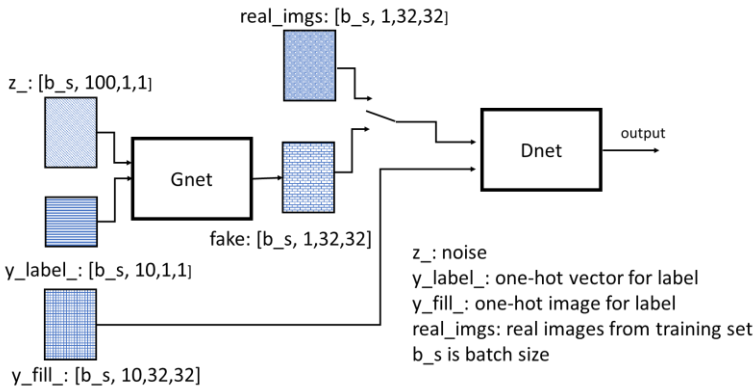


Fig.12.21 The major tensor variables in the training loop

```
# Training Loop

# Lists to monitor the training progress
G_losses = []
D_losses = []
num_epochs = 10
print("Starting Training Loop...")


for epoch in range(num_epochs):
```

```
    # For each epoch
    y_real_ = torch.ones(batch_size)
    y_fake_ = torch.zeros(batch_size)
    y_real_ = y_real_.to(device)
    y_fake_ = y_fake_.to(device)

    # y_real_= [1,1,1,1,...,1]
    # y_fake_= [0,0,0,0,...,0]

    for i, data in enumerate(dataloader, 0):

        # For each batch
        # -----------------------------------------------------------
        # (1) Update D network: maximize log(D(x))+log(1-D(G(z)))
        # -----------------------------------------------------------

        # (1) (A)Train with all-real image batch
        Dnet.zero_grad()
        real_imgs= data[0].to(device) #data[0] are images for the batch
        b_s = real_imgs.size(0)        #b_s is batch size
        label=torch.full((b_s,),real_label,dtype=torch.float,device=device)
        if b_s != batch_size:
            y_real_ = torch.ones(b_s)
            y_fake_ = torch.zeros(b_s)
            y_real_ = y_real_.to(device)
            y_fake_ = y_fake_.to(device)

        y_fill_ = fill[data[1]]  # y_fill_ : [b_s, 10, 32, 32]
                                 # data[1]: labels
        y_fill_ = y_fill_.to(device)

        # Calculate the output of D
        output = Dnet(real_imgs, y_fill_).view(-1)
        # Calculate loss on the real batch
        errD_real = criterion(output, y_real_)
        # Calculate gradients for D
        errD_real.backward()
        D_x = output.mean().item()

        # (1) (B) Train with all-fake image batch

        # Generate batch of latent vectors
        # Gnet inputs: z_ is random noise,
        #              y_label_ is random label in one-hot vector
        # Dnet inputs: fake from Gnet, and y_fill_
        z_ = torch.randn((b_s, 100)).view(-1, 100, 1, 1)
        # z_ is [b_size, 100, 1, 1]
        y_ = (torch.rand(b_s, 1) * 10).type(torch.LongTensor).squeeze()
        # y_ is [b_size] and is random labels
        y_label_ = onehot[y_]     #: [b_s, 10, 1, 1]
        y_fill_ = fill[y_].to(device)  #: [b_s, 10, 32, 32]

        z_ = z_.to(device)
        y_label_ = y_label_.to(device)
        y_fill_ = y_fill_.to(device)

        # Generate fake image batch with G
```

```
        fake = Gnet(z_, y_label_)

        # Calculate the output of D for the fake batch
        output = Dnet(fake.detach(), y_fill_).view(-1)
        # .detach() because Gnet is not updated this time,
        # no gradients for Gnet
        # Calculate D loss on the fake batch
        errD_fake = criterion(output, y_fake_)
        # Calculate the gradients on fake batch
        errD_fake.backward()
        D_G_z1 = output.mean().item()
        # Add the gradients from the real and fake batches
        errD = errD_real + errD_fake
        # Update D
        optimizerD.step()

        #-----------------------------------------
        # (2) Update G network: maximize log(D(G(z)))
        #-----------------------------------------
        Gnet.zero_grad()
        # fake labels are real (1.0) for generator cost, because G wants to
        # fool D. So we treat the fake image as real for D.
        z_ = torch.randn((b_s, 100)).view(-1, 100, 1, 1)
        y_ = (torch.rand(b_s, 1) * 10).type(torch.LongTensor).squeeze()
        y_label_ = onehot[y_]
        y_fill_ = fill[y_]

        z_ = z_.to(device)
        y_label_ = y_label_.to(device)
        y_fill_ = y_fill_.to(device)

        fake=Gnet(z_, y_label_)
       # Calculate the output of D
        output = Dnet(fake,y_fill_).view(-1)
        # Calculate G loss
        errG = criterion(output, y_real_)
        # Calculate gradients for G
        errG.backward()
        D_G_z2 = output.mean().item()
        # Update G
        optimizerG.step()

        # Output training stats

        if i % 50 == 0:
            print('[%d/%d][%d/%d]\tLoss_D: %.4f\tLoss_G: %.4f\tD(x): %.4f\tD(
                  G(z)): %.4f / %.4f'
                  % (epoch, num_epochs, i, len(dataloader),
                     errD.item(), errG.item(), D_x, D_G_z1, D_G_z2))

        # Save Losses for plotting later
        G_losses.append(errG.item())
        D_losses.append(errD.item())
```

```
Starting Training Loop...
[0/10][0/469]   Loss_D: 1.8410   Loss_G: 5.6549   D(x): 0.6279   D(G(z)): 0.6793 / 0.0439
[0/10][50/469]  Loss_D: 0.5870   Loss_G: 2.4888   D(x): 0.8298   D(G(z)): 0.3138 / 0.0910
[0/10][100/469] Loss_D: 0.7250   Loss_G: 2.2712   D(x): 0.5948   D(G(z)): 0.0342 / 0.1242
```

```
[0/10][150/469] Loss_D: 1.5258  Loss_G: 3.4039   D(x): 0.3733   D(G(z)): 0.0149 / 0.0576
[0/10][200/469] Loss_D: 1.6579  Loss_G: 3.7255   D(x): 0.7195   D(G(z)): 0.6895 / 0.0308
...
```

Plot losses

```
plt.figure(figsize=(10,5))
plt.title("Generator and Discriminator Loss During Training", fontsize=20)
plt.plot(G_losses,label="G")
plt.plot(D_losses,label="D")
plt.xlabel("iterations", fontsize=20)
plt.ylabel("Loss", fontsize=20)
plt.legend()
plt.show()
```

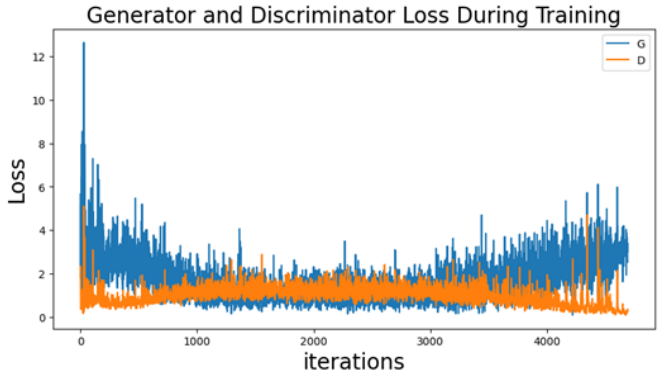


Fig.12.22 Losses of conditional GAN on MNIST for 10 training epochs

Generate fake samples. The result is plotted in Fig.12.23.

```
# this is not for training, but for generating images after training.
# generate fixed noise (fixed_z_) and fixed labels (fixed_y_label_)
# fixed_z_: 100 random vectors (z), each to generate one image
#           shape [100,100,1,1],
#           batch size =100, channel =100 (vector z), feature map:1x1
#
# fixed_y_label_: 100 one-hot vectors, each vector has 10 elements
#                 the one-hot vector for each digit repeats 10 times
#                 shape [100,10,1,1],
#                 batch size=100, channel=10 (one-hot vector)
#                 feature map: 1x1

temp_z_ = torch.randn(10, 100)
fixed_z_ = temp_z_
fixed_y_ = torch.zeros(10,1)

for i in range(9):
    fixed_z_ = torch.cat([fixed_z_, temp_z_], 0)
    temp = torch.ones(10, 1) + i
    fixed_y_ = torch.cat([fixed_y_, temp], 0)

fixed_z_=fixed_z_.view(-1,100,1,1)

fixed_y_label_ = torch.zeros(100,10)
fixed_y_label_.scatter_(1, fixed_y_.type(torch.LongTensor), 1)
```

```
fixed_y_label_ = fixed_y_label_.view(-1, 10, 1, 1)

fixed_z_ = fixed_z_.to(device)
fixed_y_label_ = fixed_y_label_.to(device)
#print(fixed_z_.shape, fixed_y_label_.shape)
```

```
with torch.no_grad():
  fake = Gnet(fixed_z_, fixed_y_label_).detach().cpu()
img = vutils.make_grid(1-fake, padding=2, nrow=10, normalize=True)

real_batch = next(iter(dataloader))

# Plot the real images
plt.figure(figsize=(15,15))
plt.subplot(1,2,1)
plt.axis("off")
plt.title("Real Images")
plt.imshow(np.transpose(vutils.make_grid(1-real_batch[0].to(device)[:100],
padding=2, nrow=10,normalize=True).cpu(),(1,2,0)))

# Plot the fake images from the last epoch
plt.subplot(1,2,2)
plt.axis("off")
plt.title("Fake Images")
plt.imshow(np.transpose(img,(1,2,0)))
plt.show()
```

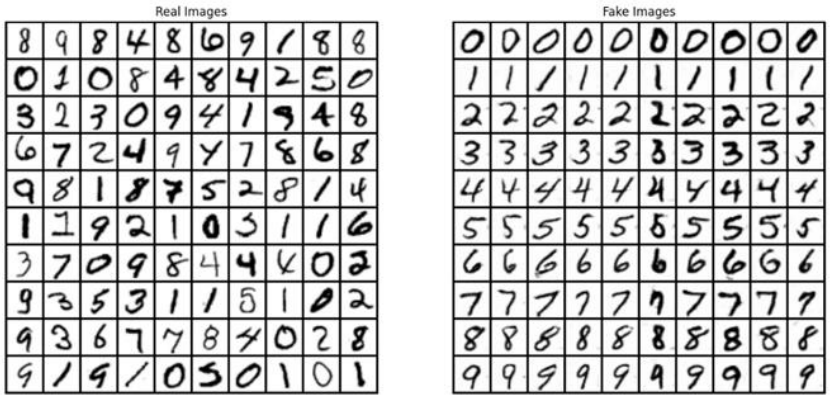


Fig.12.23 Real images vs. fake images from conditional GAN after 10 epochs training

Summary and Further Reading

Summary

Generative adversarial networks (GANs) are a kind of generative model, which are able to generate realistic high-resolution images. This chapter presents the principle of basic generative adversarial nets from the perspective of a minmax two-player game. The generator tries to generate a fake example to fool the discriminator while the discriminator tries to distinguish the fake example from the real examples. At the end of the training process, the generator and the discriminator ideally reach an equilibrium: the discriminator can hardly distinguish the fake examples from the real examples because the generator does a great job.

However, it is challenging to train a particular GAN in practice, due to the gradient vanishing and model collapse, and the sensitivity to hyperparameter selection. Various improved versions have been proposed to enhance the performance of GANs and/or the training stability. In this chapter, we present a few: conditional GAN, InfoGAN, Wasserstein GAN, CycleGAN, and f-GANs.

At the end of the chapter, a comprehensive tutorial in PyTorch is presented to implement a basic deep convolutional GAN and a conditional GAN, both on dataset MNIST.

Further Reading

The original concept of GANs was proposed by Goodfellow *et al.* 2014. A recent paper by Goodfellow *et al.* 2020 gives a brief review on the applications of GANs and identifies core research problems related to convergence necessary to make GANs a reliable technology. Deep convolutional GANs were proposed by Radford *et al.*, 2016.

To obtain more insights and details on the GAN variants in our text, one is encouraged to read the corresponding original papers. These papers include Mirza *et al*. 2014 for conditional GAN, Chen *et al.* 2016 for InfoGAN, Arjovsky *et al.* 2017 and Gulrajani *et al.* 2017 for Wasserstein GAN, Zhu *et al*., 2017 and Isola *et al*. 2017 for CycleGAN, Nowozin *et al.* 2016 for f-GANs. Some Githubs and tutorials on the Internet provide useful resources for implementations, e.g., Zeleni9, 2021 Github, S-chh 2022 Github and DCGAN PyTorch Tutorial.

There are many other GAN variants which have not been covered in our text. An interested reader may explore the following GAN variants: Laplacian GAN (Denton *et al.* 2015), Progressive GAN (Karras *et al*. 2018), Self-attention GAN (Zhang *et al*. 2019), Energy-based GAN (Zhao *et al*. 2017), Boundary equilibrium GAN (Berthelot *et al*. 2017).

GANs have been widely applied to image processing and computer vision. Some examples are given below. The super-resolution GAN by Ledig *et al*. (2017) can infer photo-realistic natural images for 4× upscaling factors. APDrawingGAN (Yi *et al*. 2019) was proposed to generate artistic portrait drawings from face photos with hierarchical GANs. There were many research efforts for face generation, such as attribute-guided face generation (Lu et al. 2018), GAN with decomposed latent spaces (Donahue et al. 2018), 3D Face Reconstruction (Gecer et al. 2019). The first effort to use GAN for video generation was the work by Vondrick et al. 2016.

File: mnist_dcgan_kuang.ipynb in Google colab. (copied to \Users\weido\ch12_gan\)
mnist_cdcgan_kuang.ipynb in Google colab. (copied to \Users\weido\ch12_gan\)

References

Ali, S.M., and Silvey, S.D. (1966), A general class of coefficients of divergence of one distribution from another. JRSS (B), pages 131–142, 1966.

Berthelot, D., Schumm, T., and Metz, L. (2017), BeGAN: Boundary equilibrium generative adversarial networks, 2017, arXiv:1703.10717 **[cs.LG]**

Chen, X., Duan, Y., Houthooft, R., Schulman, J., Sutskever, I., Abbeel, P. (2016), InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets, NIPS 2016. arXiv:1606.03657 **[cs.LG].**

DCGAN PyTorch Tutorial, https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html

Denton, E. L., Chintala, S., Szlam, A.D., and Fergus, R. (2015), Deep generative image models using a Laplacian pyramid of adversarial networks, NIPS., 2015, arXiv:1506.05751 **[cs.CV]**

Donahue, C., Lipton, Z. C., Balsubramani, A., and McAuley, J. (2018), Semantically decomposing the latent spaces of generative adversarial networks, ICLR, 2018, arXiv:1705.07904 **[cs.LG]**

Gecer, B., Ploumpis, S., Kotsia, I., and Zafeiriou, S. (2019), GANFIT: Generative adversarial network fitting for high fidelity 3D face reconstruction, CVPR, 2019, arXiv:1902.05978 **[cs.CV]**

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., Bengio, Y. (2014), Generative adversarial nets. In NIPS, 2014

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., Bengio, Y. (2020), Generative adversarial nets. pp.139-144, November 2020, Vol. 63, No. 11, Communications of the ACM.

Gulrajani, I., Ahmed, F., Arjovsky, M., Dumoulin, V., and Courville, A. (2017), Improved Training of Wasserstein GANs, NIPS 2017. arXiv:1704.00028 **[cs.LG]**

Isola, P., Zhu, J.Y., Zhou, T., and Efros, A.A. 2017, Image-to-image translation with conditional adversarial networks. In CVPR, 2017. https://arxiv.org/pdf/1611.07004.pdf

Karras, T., Aila, T., Laine, S., and Lehtinen, J. (2018), Progressive growing of GANs for improved quality, stability, and variation, ICLR, 2018, arXiv:1710.10196 **[cs.NE]**

Ledig C. et al. (2017), Photo-realistic single image super-resolution using a generative adversarial network, CVPR, 2017. arXiv:1609.04802 **[cs.CV]**

Liese, F. and Vajda, I. (2006) On Divergences and Informations in Statistics and Information Theory. IEEE Transactions on Information Theory, 52, 4394-4412.

Lu, Y., Tai, Y.W., and Tang C.K. (2018), Attribute-guided face generation using conditional cycleGAN, in Proc. Eur. Conf. Comput. Vis., 2018, pp. 282–297. arXiv:1705.09966 **[cs.CV]**

Mirza, M., and Osindero, S. (2014), Conditional Generative Adversarial Nets, 2014. arXiv:1411.1784 **[cs.LG].**

Nguyen, X., Wainwright, M. J., and Jordan, M.I. (2010), Estimating divergence functionals and the likelihood ratio by convex risk minimization. Information Theory, IEEE, 56(11):5847–5861, 2010.

Nowozin, S., Cseke, B., and Tomioka, R. (2016), f-GAN: Training generative neural samplers using variational divergence minimization, NIPS, 2016, arXiv:1606.00709 **[stat.ML]**

Radford A., Metz, L., and Chintala, S. 2016, Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks, ICLR, arXiv:1511.06434 **[cs.LG].**

S-chh, 2022, Github: https://github.com/s-chh/PyTorch-GAN-Variants

Villani, C. (2009) Optimal Transport: Old and New. Grundlehren der mathematischen Wissenschaften. Springer, Berlin, 2009. https://cedricvillani.org/sites/dev/files/old_images/2012/08/preprint-1.pdf

Vondrick, C., Pirsiavash, H., and Torralba, A. (2016), Generating videos with scene dynamics, in NIPS, 2016, arXiv:1609.02612 **[cs.CV]**

Yi, R., Liu, Y. J., Lai, Y. K., and Rosin, P. L. (2019), APDrawingGAN: Generating artistic portrait drawings from face photos with hierarchical GANs, CVPR, 2019.

Zeleni9, 2021, Github: https://github.com/Zeleni9/pytorch-wgan

Zhang, H., Goodfellow, I., Metaxas, D., and Odena, A. (2019), Self-attention generative adversarial networks, ICML, 2019, arXiv:1805.08318 **[stat.ML]**

Zhao, J., Mathieu, M., and LeCun, Y. (2017), Energy-based generative adversarial network, ICLR, 2017. arXiv:1609.03126 **[cs.LG]**

Zhu, J.Y., Park, T., Isola, P., and Efros, A.A., 2017, Unpaired Image-to-Image Translation using Cycle-consistent Adversarial Networks, ICCV, 2017. arXiv:1703.10593 **[cs.CV]**

Exercises

12.1 Show that the optimal discriminator in the original GAN by Goodfellow is given by

$$D_g^*(\mathbf{x}) = \frac{p_{data}(\mathbf{x})}{p_{data}(\mathbf{x}) + p_g(\mathbf{x})}$$

(Hint: see the original paper Goodfellow et al. 2014)

12.2 To compute the transposed convolution in Fig.12.5, we can use three different ways:

1) Equation (12.15): $Z = \overline{W}^T \times Y$.
2) Flipping the filter sheet twice in Fig.12.6.
3) Summing the shifted and weighted filters in Fig.12.7.

Please compute the transposed convolution in Fig.12.5 using the above different ways, and verify that they lead to the same result.

12.3 Consider a transposed convolution for the two-channel input. The input and the kernel are given below.

Input

1	**2**
0	**0**

0	**0**
0	**3**

kernel

0	0	0
1	1	1
0	0	0

1	0	0
0	1	0
0	0	1

Compute the transposed convolutions for the following settings (p for zero-padding, s for stride)

1) Default setting, i.e., p=0, s=1.
2) p=1, s=1.
3) p=0, s=2.
4) p=1, s=2.

Use torch.nn.ConvTranspose2d () to verify your results.

12.4 In CycleGAN (Section 12.3.5), the original authors suggested a 70×70 PatchGAN as the architecture of the discriminator. Please explain the meaning of 70.

12.5 Given a valid *f*-divergence function $f(u)$: $\mathbb{R}_+ \to \mathbb{R}$, find another function $\tilde{f}(u)$ such that

$$D_f(p||q) = D_{\tilde{f}}(q||p)$$

12.6 Find the additional constraint on a valid *f*-divergence function $f(u)$, such that the *f*-divergence is symmetric, i.e.,

$$D_f(p||q) = D_f(q||p)$$

12.7 Derive the min-max objective functions for the f-GANs based on the following *f*-divergences.

1) KL divergence.
2) Jessen-Shannon divergence.

12.8 Run the programs in Section 12.4 a few times independently. You may find out that the training process does not learn at all for some runs. What are the possible reasons?

12.9 Implement and train the following GANs:

1) InfoGAN (Table 12.2) on MNIST dataset.
2) Wasserstein GAN weight clipping (Table 12.3) on CelebA dataset.
3) Wasserstein GAN gradient penalty (Table 12.3) on CelebA dataset.
4) *f*-GAN using KL divergence on MNIST dataset. (Refer to Nowozin *et al*. 2016 or other resources).

Keys to Exercises:

12.3

1) [[0,0,0,0], [1,6,3,2], [0,0,3,0], [0,0,0,3]]
1) [[6,3], [0,3]]
2) [[0., 0., 0., 0., 0.],[1., 1., 3., 2., 2.],[0., 0., 3., 0., 0.], [0., 0., 0., 3., 0.], [0., 0., 0., 0., 3.]]
3) [[1., 3., 2.], [0., 3., 0.], [0., 0., 3.]]