Chapter 8

Generative Adversarial Networks

In previous chapters, we explored modeling high-dimensional complex data distributions using Autoregressive Models, Variational AutoEncoders (VAEs), and Flow Models. While each of these models has its advantages and disadvantages in terms of computing densities—whether exactly or through approximations, sampling efficiency, and representation learning—they all share a common characteristic: they are likelihood-based models. This means they use Maximum Likelihood Estimation (MLE) as the primary approach for learning their parameters. In the generative learning literature, these kinds of models are known as explicit models.

In this chapter, we will delve into a different approach for learning generative models, referred to as implicit models in the literature, as shown in Figure 8.1. These models do not require the ability to compute the probability density. This chapter will focus on Generative Adversarial Networks (GANs) as a prominent example of implicit models, with a training process that focuses on generating high-quality realistic high-dimensional samples rather than performing density estimation.

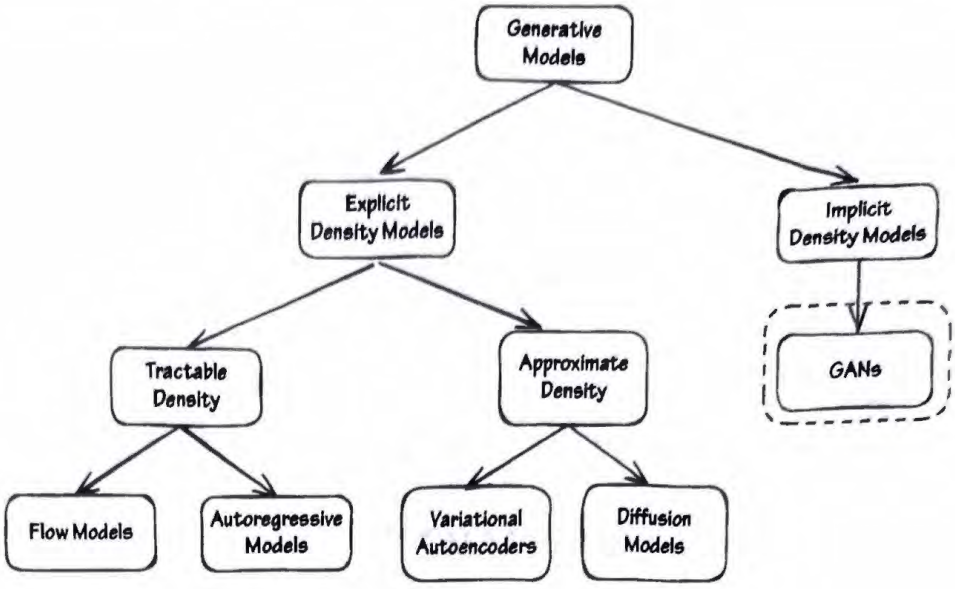

Figure 8.1 Model taxonomy: GANs.

8.1 Introduction

In previous chapters, we discussed the primary goals of generative models: to build models for high-dimensional data that can accomplish the following:

- Compute probability densities
- Perform representation learning (identifying hidden structures)
- Generate new samples

Choosing a single metric that measures the quality all these objectives can be challenging. A specific metric or training objective might be good for representation learning but not for generating high-quality samples, for example. However, intuitively, if a model can closely approximate the data distribution, it should, in theory, perform well in the other tasks. Model families like Autoregressive Models, VAEs, and Flow Models use this principle and during the learning process, try to align the model distribution as closely as possible to the data distribution. This alignment ensures that the resulting model is not only a good density estimator but also provides useful hidden structures and high-quality samples. All these model families propose a tractable or approximate density model class such that the learning can be based on Maximum Likelihood Estimation, which we saw is equivalent to minimizing the Kullback-Leibler divergence between the data distribution and the model distribution.

Generative Adversarial Networks (GANs), on the other hand, focus their training process on producing a model that can generate high-quality samples without the need to compute density estimation explicitly. The inventor of GANs, Ian Goodfellow, described his solution in his paper (Goodfellow et al., 2014), detailing how this can be achieved. His very clever idea is to use a different neural network, called the

discriminator, whose sole task is to score whether an observation sample has been generated by the data distribution or by the model distribution and to use this signal provided by the discriminator to improve the model distribution. The model distribution is called the generator in GAN terminology. Ideally, a good generator will produce samples such that one cannot distinguish between the distribution generated by the model and the actual data distribution.

Yann LeCun who received the 2018 Turing Award, together with Yoshua Bengio and Geoffrey Hinton, for their work on deep learning, declared GANs to be "the most interesting idea in the last 10 years in machine learning." As a historical note, Ian Goodfellow conceived generative adversarial networks while brainstorming programming techniques with friends at a bar. For more details, read the interview at https://www.deeplearning.ai/the-batch/ian-goodfellow-a-man-a-plan-a-gan/.

Similar to VAEs or Flow Models, in GANs, the sampling process for the generator consists of first sampling a latent variable $\mathbf{z}$ from a prior distribution $p_{\mathbf{z}}$, which is chosen to be easy to sample from, and then mapping the latent variable space to the observation or data space using the generator, denoted by $G_{\boldsymbol{\theta}}$ and parameterized by $\boldsymbol{\theta}$. This mapping can be either probabilistic or deterministic. For a deterministic mapping $G_{\boldsymbol{\theta}}$, sampling can be computed very efficiently, as in Flow Models. The sampling process in GANs is given by:

$$\begin{aligned} \mathbf{z} &\sim p(\mathbf{z}) \\ \mathbf{x} &= G_{\boldsymbol{\theta}}(\mathbf{z}) \end{aligned} \tag{8.1}$$

A key difference from Flow Models is that, because we are not using the density for model learning, the generator can be any neural network. There is no requirement that the mapping be invertible, and the dimensions of the latent variable $\mathbf{z}$ and the observations $\mathbf{x}$ can be different, as in VAEs, which can provide interesting learned representations.

Unlike Flow Models, where we train only the encoder or flow, GANs require learning the parameters of two different models simultaneously: the generator neural network $G_{\boldsymbol{\theta}}$ and the discriminator neural network, denoted by $D_{\boldsymbol{\phi}}$ and parameterized by $\boldsymbol{\phi}$.

The big-picture idea of what happens during training is the following. We train the discriminator, which is a binary classifier that classifies real data (generated by the data distribution), versus fake data (generated by the generator), optimizing its parameters to get better at discriminating these classes. To train the discriminator, we build the "real dataset," denoted by $\mathcal{D}_T$, by collecting real data related to our application (e.g., images of faces) and assigning the target label true or positive (or 1) to all of them. We build the "fake dataset," denoted by $\mathcal{D}_F$, by randomly initializing the generator $G_{\boldsymbol{\theta}}$, sampling a number of images, and assigning the target label false or negative (or 0) to all of them. With the combined dataset $\mathcal{D} = \mathcal{D}_T \cup \mathcal{D}_F$, we train the discriminator $D_{\boldsymbol{\phi}}$. Once the discriminator is trained, we use the signal provided by the discriminator as a loss function to train $G_{\boldsymbol{\theta}}$, and iterate.

Figure 8.2 Evolution of GAN-generated images over time, illustrating the progressive improvement in image generation. This figure is adapted from a tutorial by Ian Goodfellow titled "Ian Goodfellow: Adversarial Machine Learning" at ICLR 2019. The image is sourced from a screenshot taken at the 6:47 mark of the presentation. *Source:* Ian Goodfellow: Adversarial Machine Learning (ICLR 2019 invited talk) / https://www.youtube.com/watch?v=sucqskXRkss / / last accessed on January 17, 2025.

Thus, the GAN training process involves a game where the discriminator tries to get better at classifying real versus fake data, and the generator tries to get better at fooling the discriminator into classifying fake data as real. This is the adversarial part of the game, which can eventually reach an equilibrium called the Nash Equilibrium.

GANs are known for generating very high-quality high-dimensional images. In Figure 8.2, we see a progression of samples generated by GANs over a span of 4.5 years, from the original work by Goodfellow et al. (2014) in 2014 to 2018. However, GANs are known to be difficult to train. In the following sections, we will delve into more details about the training process and what GANs are actually learning. The following exposition was inspired by the style presented in Levine (2021).

8.2 Training

Training GANs involves solving a two-player game between the discriminator and the generator, as described in the original formulation by Goodfellow et al. (2014). The goal is to solve the following min-max optimization problem:

$$\min_{\theta} \max_{\phi} V(\theta, \phi) \tag{8.2}$$

where the objective function $V(\theta, \phi)$ is given by:

$$V(\theta, \phi) = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}}[\log D_{\phi}(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}}[\log(1 - D_{\phi}(G_{\theta}(\mathbf{z})))] \tag{8.3}$$

In this formulation, $V(\theta, \phi)$ represents the loss function of the discriminator D_ϕ, which takes an input and outputs the probability that it belongs to the true dataset (real observations). This loss function has the form of a typical cross-entropy loss (negative log likelihood) used in binary classification problems (see Murphy [2022]; López de Prado, 2018), where the goal in this particular case is to distinguish real from fake data. The discriminator aims to maximize this objective, while the generator aims to minimize it, creating and adversarial training dynamic. The discriminator seeks to maximize the log-likelihood of real data (positive labels) and the log-likelihood of fake data (negative labels). Thus, in the previous equation, the first expectation is taken with respect to the distribution of real data, p_{data}, and the second expectation is taken with respect to the latent variable distribution, $p_{\mathbf{z}}$. which is used to generate the fake dataset.

These expectations are approximated using sample-based estimators. The objective function can be approximated by:

$$\begin{aligned} V(\theta, \phi) &= \sum_{n=1}^{N} \log D_\phi(\mathbf{x}_n) + \sum_{m=1}^{M} \log\big(1 - D_\phi(G_\theta(\mathbf{z}_m))\big) \\ &= \sum_{n=1}^{N} \log D_\phi(\mathbf{x}_n) + \sum_{m=1}^{M} \log\big(1 - D_\phi(\mathbf{x}_m)\big) \end{aligned} \tag{8.4}$$

where $\mathbf{x}_m = G_\theta(\mathbf{z}_m)$. In most practical applications, the number of negative samples M is set equal to the number of positive samples, N, making the discriminator's binary classification problem balanced.

To solve the min-max problem, we need to find the gradients with respect to θ and ϕ of the objective function. For the discriminator, we need the gradient with respect to ϕ:

$$\nabla_\phi V(\theta, \phi) = \nabla_\phi \Big(\sum_{n=1}^{N} \log\ D_\phi(\mathbf{x}_n) + \sum_{m=1}^{M} \log\big(1 - D_\phi(\mathbf{x}_m)\big) \Big) \tag{8.5}$$

For the generator, the gradient with respect to θ is:

$$\begin{aligned} \nabla_\theta V(\theta, \phi) &= \nabla_\theta \Big(\sum_{n=1}^{N} \log D_\phi(\mathbf{x}_n) + \sum_{m=1}^{M} \log\big(1 - D_\phi(\mathbf{x}_m)\big) \Big) \\ &= \nabla_\theta \Big(\sum_{m=1}^{M} \log\big(1 - D_\phi(\mathbf{x}_m)\big) \Big) \end{aligned} \tag{8.6}$$

Once the gradients are computed, we update the parameters using stochastic gradient ascent in the discriminator, and stochastic gradient descent on the generator:

$$\begin{aligned} \phi &\leftarrow \phi + \alpha_1 \nabla_\phi V(\theta, \phi) \\ \theta &\leftarrow \theta - \alpha_2 \nabla_\theta V(\theta, \phi) \end{aligned} \tag{8.7}$$

where α_1 and α_2 are the learning rates for the discriminator and the generator, respectively.

An interesting aspect is the computation of the gradients for the generator. While the discriminator's gradient computation is straightforward, the generator's gradient requires backpropagating through the discriminator. If we denote $\ell = \log\big(1 - D_\phi(G_\theta(\mathbf{z}))\big)$, then:

$$\begin{aligned}
\nabla_\theta V(\theta, \phi) &= \nabla_\theta \Big(\sum_{m=1}^{M} \log\big(1 - D_\phi(\mathbf{x}_m)\big) \Big) \\
&= \nabla_\theta \Big(\sum_{m=1}^{M} \log(1 - D_\phi(G_\theta(\mathbf{z}_m)) \Big) \\
&= \Big(\sum_{m=1}^{M} \nabla_\theta \log(1 - D_\phi(G_\theta(\mathbf{z}_m)) \Big) \\
&= \sum_{m=1}^{M} \nabla_\theta \ell_m \\
&= \sum_{m=1}^{M} \frac{\partial \mathbf{x}_m}{\partial \theta} \frac{\partial \ell_m}{\partial \mathbf{x}_m}
\end{aligned} \tag{8.8}$$

This indicates that we are backpropagating the gradients through the discriminator to the generator. In practice, we concatenate the generator and the discriminator as a compose function, $D_\phi(G_\theta(\mathbf{z}))$, and our favorite automatic differentiation tool will compute the entire gradient efficiently.

The typical training procedure for GANs involves the following steps:

1. Obtain a true dataset $\mathcal{D}_T = \{x_n\}$ and label each observation as real (positive label).
2. Initialize the Generator G_θ randomly.
3. Generate a fake dataset, $\mathcal{D}_F = \{x_m\}$, *by sampling from the generator*, $\mathbf{z}_m \sim p_\mathbf{z}$, $x_m = G_\theta(\mathbf{z}_m)$, and label each observation as fake (negative label).
4. Combine the true and fake datasets to create a combined dataset $\mathcal{D} = \mathcal{D}_T \cup \mathcal{D}_F$.
5. Optimize the discriminator $D_\phi(\mathbf{x}) = p(\mathbf{y} \mid \mathbf{x})$ using the combined dataset $\mathcal{D}$ with one step of stochastic gradient ascent.
6. Optimize the generator D_θ using the feedback signal from the discriminator with one step of stochastic gradient descent.
7. Repeat steps 3 to 6 until convergence.

By alternative updating the discriminator and the generator, we train the GAN to improve the quality of the generated data, making it increasingly difficult for the discriminator to distinguish between real and fake data more difficult.

In Figure 8.3, we show samples generated as illustrated in the original work by Goodfellow et al. (2014). At that time, the sample quality was at least competitive with the best generative models available, as noted by the authors. In Figure 8.3, the rightmost column shows the nearest training example to the neighboring sample, demonstrating that the model has not memorized the training set. This is illustrated for the following datasets: (a) MNIST, (b) TFD, (c) CIFAR-10 (fully connected model), and (d) CIFAR-10 (convolutional discriminator and "deconvolutional" generator). For more details, see Goodfellow et al. (2014).

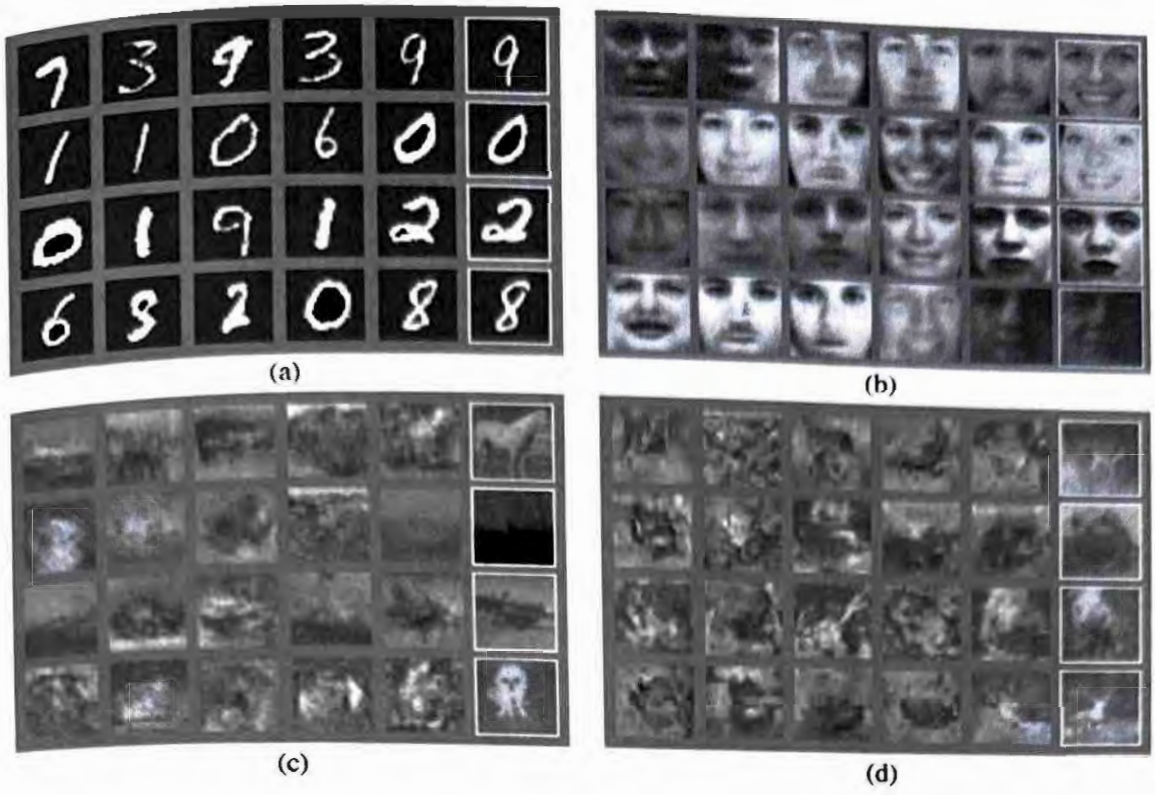

Figure 8.3 The rightmost column displays the nearest training example to each corresponding generated sample, demonstrating that the model has not memorized the training set. This is illustrated for the following datasets: (a) MNIST, (b) TFD, (c) CIFAR-10 (fully connected model), and (d) CIFAR-10 (convolutional discriminator and "deconvolutional" generator). *Source:* I. J. Goodfellow et al. (2014).

Model	MNIST	TFD
DBN [3]	138 ± 2	1909 ± 66
Stacked CAE [3]	121 ± 1.6	$\mathbf{2110 \pm 50}$
Deep GSN [6]	214 ± 1.1	1890 ± 29
Adversarial nets	$\mathbf{225 \pm 2}$	$\mathbf{2057 \pm 26}$

Figure 8.4 Table 1 from Goodfellow et al. (2014): Parzen window-based log-likelihood estimates. The reported values for MNIST represent the mean log-likelihood of test set samples, with the standard error of the mean computed across individual examples.

8.2.1 Evaluation

The evaluation of GANs is still an open problem. Since we cannot compute the density explicitly, and the discriminator probability score is used to drive the generator to create data that are similar to the data distribution, this score for a given observation does not provide much insight into the quality of the generated samples. This is especially important when we want to compare models. One initial solution was to use Parzen window-based log-likelihood estimates, also known as Kernel Density Estimation (KDE), as a replacement for likelihoods. More details about KDE can be found in Bishop (2006).

The Figure 8.4 shows the Parzen window-based log-likelihood on the original work on GANs, allowing for a quantifiable model comparison rather than simply eyeballing the generated images. They reported numbers on MNIST and TFD datasets for different generated models.

More evaluation metrics have been proposed in the literature to better account for the coverage of the generated samples. One such metric is the Inception Score, see Salimans et al. (2016), which utilizes the Inception neural network to classify images and evaluate the quality of conditional generation with GANs. Improvements over the Inception Score include the Frechet Inception Distance (FID), which aims to improve the measurement of diversity by incorporating statistics of the images in a different embedding space. More details about the FID distance can be found in Heusel et al. (2018).

8.3 Some Theoretical Insight in GANs

We have already seen that the training of GANs requires solving a min-max two-player game between the generator and the discriminator. From the objective function, the task for each model is clear: the discriminator maximizes the likelihood of the data, while the generator tries to fool the discriminator by minimizing the same objective function. However, since we are interested in the quality of the generated samples, we can ask ourselves the following question: Are GANs trained with the aforementioned procedure trying to optimize some kind of divergence between the data distribution p_{data} and the generator distribution p_G?

To answer this, let's perform a theoretical exercise to solve for the optimal discriminator and generator and see if we can gain some insights. Although we do not train GANs using the exact procedure to be described here, this exercise provides valuable insights that have been used in practice to improve the training and performance of GANs.

Let's solve for the optimal classifier. Given our objective:

$$\begin{aligned} &\min_{\theta}\max_{\phi} \quad V(\theta,\phi) \\ &\min_{\theta}\max_{\phi} \quad \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}\left[\log D_{\phi}(\mathbf{x})\right] + \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}}\left[\log\left(1 - D_{\phi}(G_{\theta}(\mathbf{z}))\right)\right] \end{aligned} \tag{8.9}$$

what can we say about G at convergence? Let's see if we can find a closed-form solution for D. We can rewrite Equation 8.9 as follows:

$$\begin{aligned} &\min_{G}\max_{D} \quad \mathbb{E}_{\mathbf{x}\sim p}\left[\log D(\mathbf{x})\right] + \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}}\left[\log(1 - D(G(\mathbf{z})))\right] \\ &\min_{G}\max_{D} \quad \mathbb{E}_{\mathbf{x}\sim p}\left[\log D(\mathbf{x})\right] + \mathbb{E}_{\mathbf{x}\sim q}\left[\log(1 - D(\mathbf{x}))\right] \end{aligned} \tag{8.10}$$

Here, we simplify our notation and optimize over functions D and G instead of optimizing over parameters θ and ϕ. We also replace p_{data} by p, the distribution of the true dataset, and let q represent the distribution used to generate our fake dataset, with $D(\mathbf{x}) = D(G(\mathbf{z}))$.

Let's solve for the optimal discriminator D:

$$D^{*} = \arg\max_{G} \mathbb{E}_{\mathbf{x}\sim p}\left[\log D(\mathbf{x})\right] + \mathbb{E}_{\mathbf{x}\sim q}\left[\log(1 - D(\mathbf{x}))\right] \tag{8.11}$$

To find the optimal value of D, we compute the gradient with respect to D and set it equal to zero:

$$\begin{aligned}\nabla_D &= \mathbb{E}_{\mathbf{x}\sim p}[\nabla_D \log D(\mathbf{x})] + \mathbb{E}_{\mathbf{x}\sim q}[\nabla_D \log(1 - D(\mathbf{x}))] \\ &= \mathbb{E}_{\mathbf{x}\sim p}\left[\frac{1}{D(\mathbf{x})}\right] - \mathbb{E}_{\mathbf{x}\sim q}\left[\frac{1}{1 - D(\mathbf{x})}\right]\end{aligned} \tag{8.12}$$

Setting $\nabla_D = 0$ and solving for D, we find that optimal value is:

$$\begin{aligned}D^* &= \frac{p(x)}{p(x) + q(x)} \\ D^* &= \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_G(x)}\end{aligned} \tag{8.13}$$

where q is replaced by p_G, to indicate that the negative labels are generated from the generator distribution, p_G.

Now, if we plug the optimal value of D into the objective function, we can determine what the generator G is trying to optimize:

$$\begin{aligned}V(G, D^*) &= \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}[\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim p_G}[\log(1 - D(\mathbf{x}))] \\ &= \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}\left[\log \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_G(x)}\right] + \mathbb{E}_{\mathbf{z}\sim p_G}\left[\log \frac{p_G(x)}{p_{\text{data}}(x) + p_G(x)}\right] \\ &= \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}[\log p_{\text{data}}(x) - \log(p_{\text{data}}(x) + p_G(x))] + \\ &\quad \mathbb{E}_{\mathbf{x}\sim p_G}[\log p_G(x) - \log(p_{\text{data}}(x) + p_G(x))]\end{aligned} \tag{8.14}$$

Let $q(x) = (p_{\text{data}}(x) + p_G(x))/2$. Then

$$\begin{aligned}V(G, D^*) &= \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}[\log p_{\text{data}}(x) - \log q(x)] + \mathbb{E}_{\mathbf{x}\sim p_G}[\log p_G(x) - \log q(x)] - \log 4 \\ &= D_{\text{KL}}(p_{\text{data}} \| p_G) + D_{\text{KL}}(p_G \| p_{\text{data}}) - \log 4 \\ &= 2D_{\text{JS}}(p_{\text{data}} \| p_G) - \log 4\end{aligned} \tag{8.15}$$

where D_{JS} is the Jensen-Shannon Divergence.

Thus, at convergence, the generator minimizes a notion of distance between the data distribution p_{data} and the generator distribution p_G. This insight has led to modern approaches that address both the drawbacks and advantages of minimizing this divergence, proposing better alternatives for training GANs.

8.4 Why Is GAN Training Hard? Improving GAN Training Techniques

Training GANs in practice is challenging. Implementing GANs as described in the original work by Goodfellow et al. (2014) requires an enormous amount of hyperparameter tuning to achieve convergence and generate high-quality samples.

For example, the paper by Salimans et al. (2016) illustrates the extensive hyperparameter tuning required and some of the techniques used to make GANs work effectively. These techniques include minibatch discrimination, feature matching,

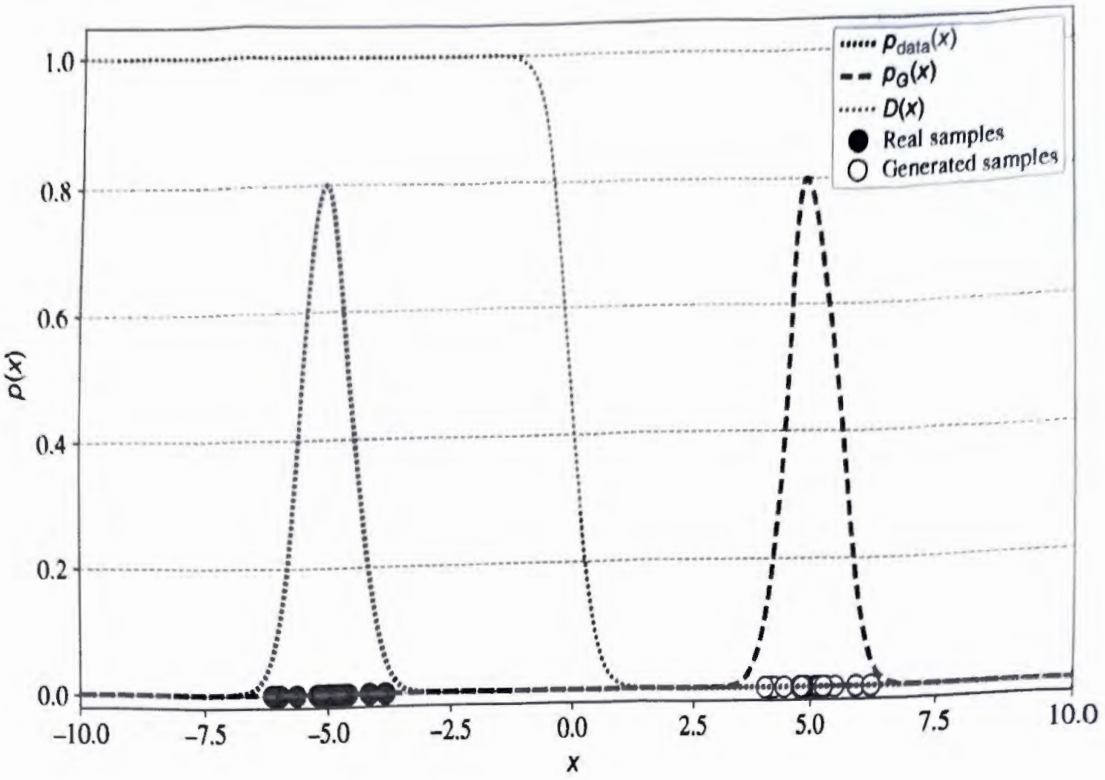

Figure 8.5 Illustration of one reason why training GANs can be difficult. In this conceptual illustration, at initialization, the data distribution and the generator distribution are far apart, allowing the discriminator to achieve perfect classification. This results in a sharp decision boundary, where the discriminator outputs 1 for real samples and 0 for generated samples, making it challenging for the generator to receive meaningful gradient updates.

historical averaging, and others, which are applicable to either the generator, the discriminator, or both. For more details, see Salimans et al. (2016).

One reason why training GANs is difficult can be understood by examining a simple example, illustrated in Figure 8.5. For the sake of simplicity, we generate samples from both the data distribution and the generator, overlaying the corresponding density functions and assuming this is the initial data for training. At initialization, with a random generator, the densities and samples generated from p_{data} and p_G are far apart. If we train the discriminator with the combined dataset $\mathcal{D} = \mathcal{D}_T \cup \mathcal{D}_F$ using the true data $\mathcal{D}_T$ and the fake data $\mathcal{D}_F$ from these samples, the discriminator can easily achieve perfect classification. This results in a very sharp decision boundary, saturated at 1 for samples from p_{data} and 0 for samples generated by p_G.

As the generator learns by backpropagating the complete gradients of the discriminator, in situations where the discriminator is saturated, it provides very poor gradient signals to the generator for almost all the points that are generated from p_G. This makes it almost impossible for the generator to improve the quality of the generated samples. In short, one of the main reasons why GAN training is hard is that the discriminator might provide a very weak gradient signal, if the data distribution and the generator distribution are very far apart.

To improve this situation where p_{data} and p_G are very far apart, we aim to have a smoother decision boundary for the discriminator. This would provide meaningful signals to drive the training of the generator and improve the quality of the generated samples. Solutions around this point include explicitly modifying the discriminator to have a smoother decision boundary or making the samples from p_{data} and p_G overlap, making it more difficult for the discriminator to classify them. Instance noise, as described by Mescheder et al. (2018), is a method where noise is added to both real and fake samples to make their distributions overlap, making the classification task harder for the discriminator and resulting in a smoother decision boundary.

Techniques for explicitly modifying the decision boundary include Least Squares Generative Adversarial Networks (LSGANs), as described by Mao et al. (2017), where the discriminator outputs an unbounded real number instead of a probability, a real number between 0 and 1. Other solutions involve using different objective functions to force the discriminator to have a smoother decision boundary. The main solution proposed in Wasserstein GANs (WGANs), as described in Arjovsky et al. (2017), constrains the discriminator to be Lipschitz-continuous. Further solutions for making the decision boundary even smoother include WGAN with Gradient Penalty (WGAN-GP) by Gulrajani et al. (2017) and Spectral Normalization GAN (SNGAN) by Miyato et al. (2018).

For detailed analysis of convergence and different training techniques for GANs, refer to the paper by Mescheder et al. (2018).

8.5 Wasserstein GAN (WGAN)

From Section 8.3, we got the intuition that the generator is trying to minimize the Jensen-Shannon (JS) divergence between the data distribution and the generator distribution. However, the JS divergence is not effective at capturing the distance between distributions when they are very far apart or have little to no overlap.

To illustrate this, consider Figure 8.6. In the top figure, the distributions p_{data} and p_G are very far apart with zero overlap. In the bottom figure, p_{data} and p_G are closer but still do not overlap. Ideally, we want to drive the generator distribution p_G closer to p_{data}, and between these two options, we would prefer the generator from the bottom figure. However, the JS divergence assigns the same score to both situations since there is no overlap between the two distributions, which poses a problem for training the generator effectively.

An improvement to this situation could be to use a different divergence measure that actually quantifies the distance between two distributions. One candidate is the Wasserstein distance, which is used to optimize GANs as proposed by Arjovsky et al. (2017).

The Wasserstein distance is defined as:

$$W(p_{\text{data}}, p_G) = \inf_{\gamma} \mathbb{E}_{(\mathbf{x},\mathbf{y})\sim\gamma} \|\mathbf{x} - \mathbf{y}\| \tag{8.16}$$

where γ is the distribution over $\mathbf{x}$ and $\mathbf{y}$ that satisfies the marginals $\gamma_{\mathbf{x}} = p_{\text{data}}$ and $\gamma_{\mathbf{y}} = p_G$.

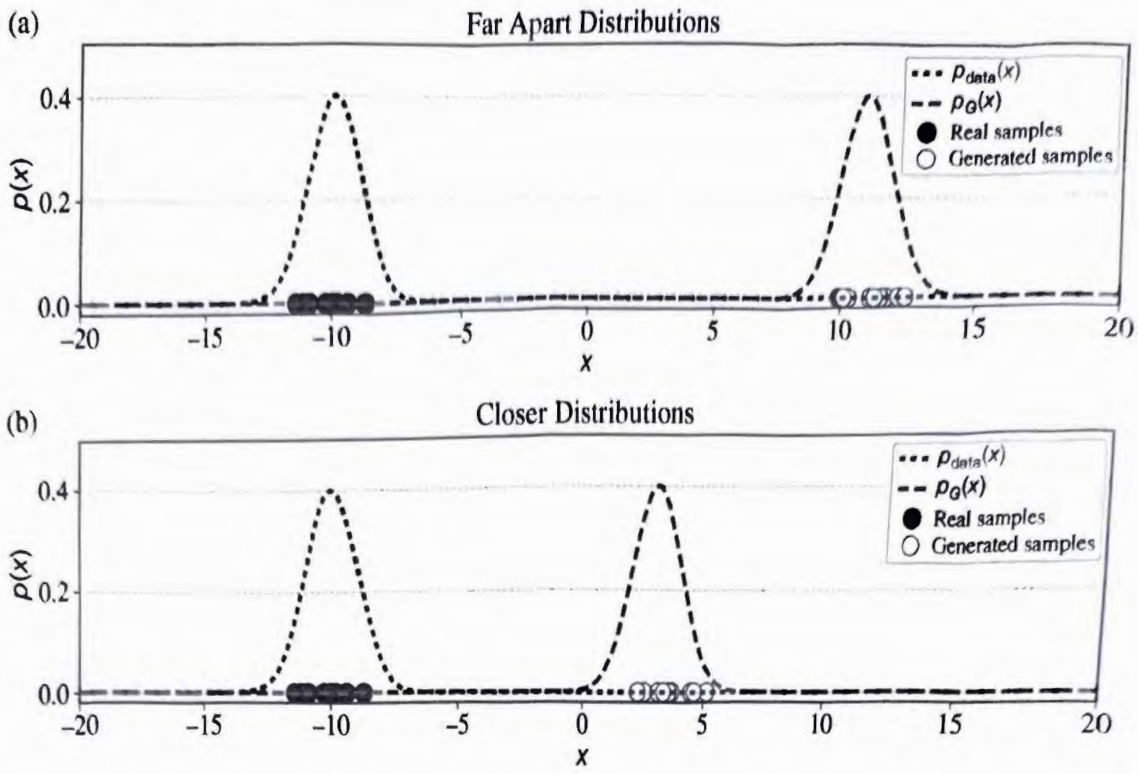

Figure 8.6 Illustration of two different cases for the data distribution and the generator distribution. In Figure (a), the distributions are far apart, whereas in Figure (b), they are closer. Despite this difference, the JS divergence assigns the same score to both cases.

Directly optimizing this formulation of $W(p_{\text{data}}, p_G)$ is complicated because finding the optimal γ is not an easy task. However, there is a dual formulation known as the Kantorovich-Rubinstein duality, which makes the problem more tractable:

$$W(p_{\text{data}}, p_G) = \sup_{\|f\|_L \leq 1} \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}} f(\mathbf{x}) - \mathbb{E}_{\mathbf{x} \sim p_G} f(\mathbf{x}) \tag{8.17}$$

This is equal to the supremum over the set of all possible 1-Lipschitz functions f, of the expected value of f under p_{data} minus the expected value of f under p_G. The set of all possible 1-Lipschitz scalar functions f satisfies:

$$|f(x) - f(y)| \leq |x - y| \tag{8.18}$$

which means roughly speaking that f should have a bounded gradient or slope.

Returning to our example, if f_ϕ is our discriminator neural network parameterized by ϕ, how can we constrain the network to be 1-Lipschitz, or equivalently, to have a bounded gradient? The original solution proposed by Arjovsky et al. (2017) involves clipping the weights of the network to ensure the gradient is bounded. However, the paper's authors clearly mentioned that weight clipping is probably not the best way to enforce this constraint and illustrated many problems with it. In fact, at the time of writing, gradient clipping for achieving this constraint is mostly obsolete. Other works build on this idea

to enforce a Lipschitz-continuous neural network, such as WGAN with Gradient Penalty (WGAN-GP) by Gulrajani et al. (2017) and Spectral Normalization GAN (SNGAN) by Miyato et al. (2018).

Once we constrain the search space of functions to the set of 1-Lipschitz functions, we can use the sample-based objective to find the optimal discriminator and generator:

$$\begin{aligned} W(p_{\text{data}}, p_G) &= \sup_{\|f_\phi\|_L \le 1} \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}[f_\phi(\mathbf{x})] - \mathbb{E}_{\mathbf{x}\sim p_G}[f_\phi(\mathbf{x})] \\ &= \sup_{\|f_\phi\|_L \le 1} \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}[f_\phi(\mathbf{x})] - \mathbb{E}_{\mathbf{z}\sim p_z}[f_\phi(G_\theta(\mathbf{z}))] \end{aligned} \tag{8.19}$$

We approximate $W(p_{\text{data}}, p_G)$ using sample-based estimators:

$$\widehat{W}(p_{\text{data}}, p_G) = \sup_{\|f_\phi\|_L \le 1} \sum_{n=1}^{N} f_\phi(\mathbf{x}_n) - \sum_{m=1}^{M} f_\phi(G_\theta(\mathbf{z}_m)) \tag{8.20}$$

We can then optimize by taking gradients with respect to ϕ and θ.

The pseudo code for training WGANs is as follows (adapted from Levine (2021)):

1. Obtain a true dataset $\mathcal{D}_T = \{x_n\}$ and label each observation as real (positive label).
2. Initialize the Generator G_θ randomly.
3. Generate a fake dataset, $\mathcal{D}_F = \{x_m\}$, *by sampling from the generator*, $\mathbf{z}_m \sim p_z$, $x_m = G_\theta(\mathbf{z}_m)$, and label each observation as fake (negative label).
4. Combine the true and fake datasets to create a combined dataset $\mathcal{D} = \mathcal{D}_T \cup \mathcal{D}_F$.
5. Update the discriminator f_ϕ to maximize $\widehat{W}(p_{\text{data}}, p_G)$. Compute $\nabla_\phi \widehat{W}(p_{\text{data}}, p_G)$ and use stochastic gradient ascent.
6. Clip all weights matrices in ϕ such that each weight matrix layer, denoted by ϕ_l, is within $[-c, c]$ for some constant c.
7. Update the generator G_θ to minimize $\widehat{W}(p_{\text{data}}, p_G)$. Compute the gradients $\nabla_\theta \widehat{W}(p_{\text{data}}, p_G) = \nabla_\theta \sum_{m=1}^{M} f_\phi(G_\theta(\mathbf{z}_m))$ and use stochastic gradient descent.
8. Repeat steps 3 to 7 until convergence.

In Figure 8.7, we see the results from Figure 2 in the paper by Arjovsky et al. (2017). This figure illustrates the effect of the Wasserstein distance on optimizing GANs, particularly highlighting the decision boundary obtained by the discriminator. In Figure 8.7, the decision boundary of the discriminator for typical vanilla GANs is labeled "GAN Discriminator" in red, optimized using the two-player min-max game. The decision boundary for the WGAN, labeled "WGAN Critic" in cyan, shows a much smoother boundary, providing better gradient signals for the generator and improving the training process.

8.5.1 *Gradient Penalty GAN (WGAN-GP)*

As mentioned, another way to enforce Lipschitz constraints in a neural network is to directly constrain the gradient, which is the approach proposed in WGAN with Gradient Penalty (WGAN-GP) by Gulrajani et al. (2017).

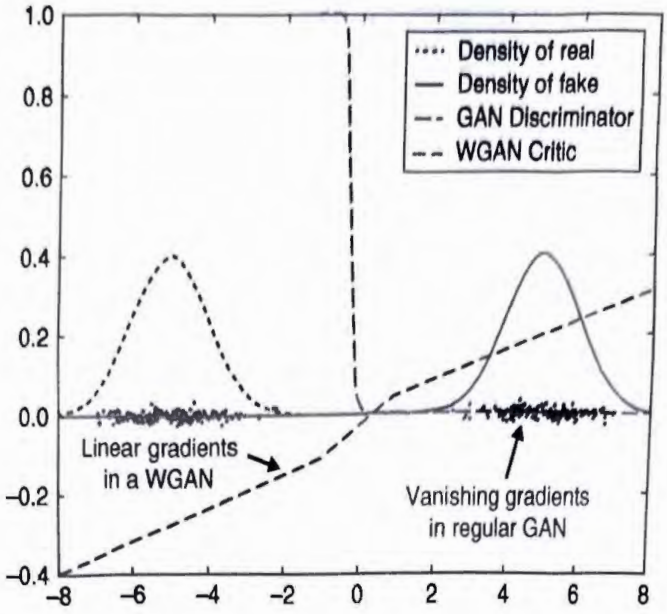

Figure 8.7 Illustration of the optimal discriminator and critic when distinguishing two Gaussians. The minimax GAN discriminator (red) saturates, leading to vanishing gradients, while the WGAN critic (cyan) provides smoother gradients, improving generator training. Figure 2 from Arjovsky, Chintala, and Bottou (2017).

In their work, they modified the original WGAN objective by introducing a gradient penalty for Lipschitz continuity, given by:

$$W(p_{\text{data}}, p_G) = \sup_{\|f_\phi\|_L \leq 1} \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}\left[f_\phi(\mathbf{x})\right] - \mathbb{E}_{\mathbf{x}\sim p_G}\left[f_\phi(\mathbf{x})\right] - \lambda \mathbb{E}_{\hat{\mathbf{x}}\sim p_{\hat{\mathbf{x}}}}\left[\left(\left\| \nabla_{\hat{\mathbf{x}}} f_\phi(\hat{\mathbf{x}}) \right\|_2 - 1 \right)^2 \right] \tag{8.21}$$

where the gradient penalty term is the last term of the previous equation. Here, the objective function directly penalizes the discriminator for having gradients with norms different from one, using the regularization parameter λ. The expectation is computed under the distribution $p_{\hat{\mathbf{x}}}$, for which samples are generated by sampling uniformly along straight lines between pairs of points sampled from the data distribution p_{data} and the generator distribution p_G. For more details, see Gulrajani et al. (2017). In practice, WGAN-GP improves the stability of WGAN training and is very easy to implement.

A detailed analysis of the convergence of different GAN training algorithms can be found in Mescheder et al. (2018).

8.6 Extending GANs for Time Series

Similarly, as we have seen in previous chapters with models like VAEs, and Flows, these models, by their structural design, assume independence between consecutive observations. Then, the natural questions arise, how can we adapt them to capture this time-series dynamics that might be important at certain time scales?

One potential answer is similar to the way we have done it previous chapters. We could try to combine GANs with models that excel at handling the kind of dynamics we want to capture, such as autoregressive models like RNN or Transformers.

Let's consider two interesting cases for time-series generation.

- Recurrent Conditional GAN (RCGAN), see Esteban et al. (2017): Uses a combination of RNN with GAN. It uses an RNN for the generator and discriminators. Due to this design choice, data are generated sequentially. More details about model architecture and experimental results are in [*RCGAN Notebook*].
- Time-series Generative Adversarial Networks (TimeGAN), see Yoon et al. (2019): TimeGAN employs a sequence generator and discriminator to model time-series dynamics. However, it does that by modeling the temporal dependencies in the latent space. They do that by combining supervised learning objectives as well as generative adversarial objectives in a jointly optimization problem. The supervised learning objective drives the network to capture the dynamics in the latent space. The adversarial objective, like in standard GANs, drives the network to generate realistic samples. More details about model architecture and experimental results are in [*TimeGAN Notebook*].

8.7 Conclusion

In this chapter, we explored another powerful tool for modeling high-dimensional data—GANs. Unlike the models studied in previous chapters, GANs belong to the class of implicit generative models, meaning they do not require explicit density estimation. Instead, they rely on an adversarial learning process in which two models—the generator and the discriminator—are trained simultaneously in an adversarial framework. Unlike flow models, the GAN generator can be any neural network, with no constraints on invertibility or matching dimensionality between the latent variable and observations. This flexibility allows GANs to learn complex representations and generate high-quality samples. We also discussed key challenges in training GANs and explored strategies to improve their stability and performance.

Additionally, we examined how GANs can be extended to model sequential data, particularly time series. By integrating GANs with architectures specialized for capturing temporal dependencies—such as RNNs—we can generate sequential data with time-dependent dynamics. In the notebooks associated with this chapter, you will find practical examples of both traditional GANs and time-series-adapted GANs applied to financial data generation.

With this chapter, you have added yet another powerful tool to your generative modeling toolbox. GANs provide a flexible and effective approach for generating high-dimensional, multimodal data.