Chapter 7
Generative Adversarial Networks

7.1 Introduction

Once we discussed latent variable models, we claimed that they naturally define a generative process by first sampling latents $\mathbf{z} \sim p(\mathbf{z})$ and then generating observables $\mathbf{x} \sim p_\theta(\mathbf{x}|\mathbf{z})$. That is nice! However, the problem appears when we start thinking about training. To be more precise, the training objective is an issue. Why? Well, the probability theory tells us to *get rid of* all unobserved random variables by marginalizing them out. In the case of latent variable models, this is equivalent to calculating the (marginal) log-likelihood function in the following form:

$$\log p_\theta(\mathbf{x}) = \log \int p_\theta(\mathbf{x}|\mathbf{z})\ p(\mathbf{z})\ \mathrm{d}\mathbf{z}. \tag{7.1}$$

As we mentioned already in the section about VAEs (see Sect. 4.3), the problematic part is calculating the integral because it is not analytically tractable unless all distributions are Gaussian and the dependency between $\mathbf{x}$ and $\mathbf{z}$ is linear. However, let us forget for a moment about all these issues and take a look at what we can do here. First, we can approximate the integral using Monte Carlo samples from the prior $p(\mathbf{z})$ that yields

$$\log p_\theta(\mathbf{x}) = \log \int p_\theta(\mathbf{x}|\mathbf{z})\ p(\mathbf{z})\ \mathrm{d}\mathbf{z} \tag{7.2}$$

$$\approx \log \frac{1}{S} \sum_{s=1}^{S} p_\theta(\mathbf{x}|\mathbf{z}_s) \tag{7.3}$$

$$= \log \sum_{s=1}^{S} \exp\left(\log p_\theta(\mathbf{x}|\mathbf{z}_s)\right) - \log S \tag{7.4}$$

J. M. Tomczak, *Deep Generative Modeling*,
https://doi.org/10.1007/978-3-030-93158-2_7

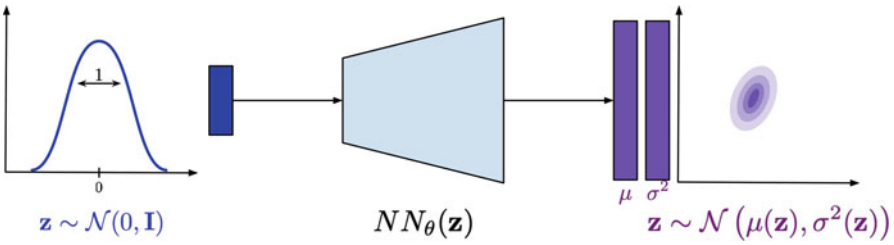

Fig. 7.1 A schematic representation of a density net

$$= \text{LogSumExp}_s \left\{ p_\theta(\mathbf{x}|\mathbf{z}_s) \right\} - \log S, \tag{7.5}$$

where $\text{LogSumExp}_s \{f(s)\} = \log \sum_{s=1}^{S} \exp(f(s))$ is the log-sum-exp function.

Assuming for a second that this is a good (i.e., a tight) approximation, we turn the problem of calculating the integral into a problem of sampling from the prior. For simplicity, we can assume a prior that is relatively easy to be sampled from, e.g., the standard Gaussian, $p(\mathbf{z}) = \mathcal{N}(\mathbf{z}|0, \mathbf{I})$. In other words, we need to model $p_\theta(\mathbf{x}|\mathbf{z})$ only, i.e., pick a parameterization for it. Guess what, we will use a neural network again! If we model images, then we can use the categorical distribution for the conditional likelihood and then a neural network parameterizes the probabilities. Or, if we use a Gaussian distribution like in the case of energy-based models or diffusion-based deep generative models, then $p_\theta(\mathbf{x}|\mathbf{z})$ could be Gaussian as well and the neural network outputs the variance and/or the mean. Since the log-sum-exp function is differentiable (and the application of the log-sum-exp trick makes it even numerically stable), there is no problem learning this model end-to-end! This approach is a precursor of many deep generative models and was dubbed *density networks* [1]; see Fig. 7.1 for a schematic representation of density networks.

Density networks are important and, unfortunately, underappreciated deep generative models. It is worth knowing them at least for three reasons. First, understanding how they work helps a lot to comprehend other latent variable models and how to improve them. Second, they serve as a great starting point for understanding the difference between *prescribed models* and *implicit models*. Third, they allow us to formulate a non-linear latent variable model and train it using backpropagation (or a gradient descent, in general).

Alright, so now you may have some questions because we made a few assumptions on the way that might have been pretty confusing. The main assumptions made here are the following:

- We need to specify the prior distribution $p(\mathbf{z})$, e.g., the standard Gaussian.
- We need to specify the form of the conditional likelihood $p(\mathbf{x}|\mathbf{z})$. Typically, people use the Gaussian distribution or a mixture of Gaussians. Hence, density nets are the *prescribed* models because we need to analytically formulate all distributions in advance.

As a result, we get the following:

- The objective function is the (approximated) log-likelihood function.
- We can optimize the objective using gradient-based optimization methods and the autograd tools.
- We can parameterize the conditional likelihood using deep neural networks.

However, we pay a great price for all the goodies coming from the formulation of the density networks:

- There is no analytical solution (except the case equivalent to the probabilistic PCA).
- We get an approximation of the log-likelihood function.
- We need a lot of samples from the prior to get a reliable approximation of the log-likelihood function.
- It suffers from the curse of dimensionality.

As you can see, the issue with dimensionality is especially limiting. What can we do with a model if it cannot be efficient for higher-dimensional problems? All interesting applications like image or audio analysis/synthesis are gone! So what can we do then? One possible direction is to stick to the prescribed models and apply variational inference (see Sect. 4.3). However, the other direction is to abandon the likelihood-based approach. I know, it sounds ridiculous, but it is possible and, *unfortunately*, works pretty well in practice.

7.2 Implicit Modeling with Generative Adversarial Networks (GANs)

Getting Rid of Kullback–Leibler

Let us think again what density networks tell us. First of all, they define a nice generative process: First sample latents and then generate observables. Clear! Then, for training, they use the (marginal) log-likelihood function. In other words, the log-likelihood function assesses the difference between a training datum and a generated object. To be even more precise, we first pick the specific probability distribution for the conditional likelihood $p_\theta(\mathbf{x}|\mathbf{z})$ that defines how to calculate the difference between the training point and the generated observables.

One may ask here whether there is a different fashion of calculating the *difference* between real data and generated objects. If we recall our considerations about hierarchical VAEs (see Sect. 4.5.2), learning of the likelihood-based models is equivalent to optimizing the Kullback-Leibler (KL) divergence between the empirical distribution and the model, $KL\left[p_{data}(\mathbf{x})||p_\theta(\mathbf{x})\right]$. The KL-based approach requires a well-behaved distribution because of the logarithms. Moreover, we can think of it as a *local* way of comparing the empirical distribution (i.e., given data) and the generated data (i.e., data generated by our prescribed model). By *local*, we

mean considering one point at a time and then summing all individual errors instead of comparing samples (i.e., collections of individuals) that we can refer to as a *global* comparison. However, we do not need to stick to the KL divergence! Instead, we can use other metrics that look at a set of points (i.e., distributions represented by a set of points) like integral probability metrics [2] (e.g., the Maximum Mean Discrepancy [MMD] [3]) or use other divergences [4].

Still, all of the mentioned metrics rely on defining explicitly how we measure the error. The question is whether we can parameterize our loss function and learn it alongside our model. Since we talk all the time about neural networks, can we go even further and utilize a neural network to calculate differences?

Getting Rid of Prescribed Distributions

Alright, we agreed on the fact that the KL divergence is only one of many possible loss functions. Moreover, we asked ourselves whether we can use a learnable loss function. However, there is also one question floating in the air, namely, do we need to use the prescribed models in the first place? The reasoning is the following. Since we know that density networks take noise and turn them into distribution in the observable space, do we really need to output a full distribution? What if we return a single point? In other words, what if we define the conditional likelihood as Dirac's delta:

$$p_\theta(\mathbf{x}|\mathbf{z}) = \delta\left(\mathbf{x} - NN_\theta(\mathbf{z})\right). \tag{7.6}$$

This is equivalent to saying that instead of a Gaussian (i.e., a mean and a variance), $NN_\theta(\mathbf{z})$ outputs the mean only. Interestingly, if we consider the marginal distribution over $\mathbf{x}$'s, we get nicely behaved distribution. To see that, let us first calculate the marginal distribution:

$$p_\theta(\mathbf{x}) = \int \delta\left(\mathbf{x} - NN_\theta(\mathbf{z})\right)\ p(\mathbf{z})\ \mathrm{d}\mathbf{z}. \tag{7.7}$$

Then, let us understand what is going on! The marginal distribution is an infinite mixture of delta peaks. In other words, we take a single $\mathbf{z}$ and plot a peak (or a point in 2D, it is easier to imagine) in the observable space. We proceed to infinity and once we do that, the observable space will be covered by more and more points and some regions will be *denser* than the others. This kind of modeling a distribution is also known as *implicit modeling*.

So where is the problem then? Well, the problem in the prescribed modeling setting is that the term $\log \delta\left(\mathbf{x} - NN_\theta(\mathbf{z})\right)$ is ill-defined and cannot be used in many probability measures, including the KL-term, because we cannot calculate the loss function. Therefore, we can ask ourselves whether we can define our own loss function, perhaps? And, even more, parameterize it with neural networks! You must admit it sounds appealing! So how to accomplish that?

Adversarial Loss

Let us start with the following story. There is a con artist (a fraud) and a friend of the fraud (an expert) who knows a little about art. Moreover, there is a real artist who has passed away (e.g., Pablo Picasso). The fraud tries to mimic the style of Pablo Picasso as well as possible. The friend expert browses for paintings of Picasso and compares them to the paintings provided by the fraud. Hence, the fraud tries to fool his friend, while the expert tries to distinguish real paintings of Picasso from fakes. Over time, the fraud becomes better and better and the expert also learns how to decide whether a given painting is a fake. Eventually, and unfortunately to the world of art, work of the fraud may become indistinguishable from Picasso and the expert may be completely uncertain about the paintings and whether they are fakes.

Now, let us formalize this wicked game. We call the expert a *discriminator* that takes an object $\mathbf{x}$ and returns a probability whether it is *real* (i.e., coming from the empirical distribution), $D_\alpha : \mathcal{X} \to [0, 1]$. We refer to the fraud as a *generator* that takes noise and turns it into an object $\mathbf{x}$, $G_\beta : \mathcal{Z} \to \mathcal{X}$. All $\mathbf{x}$'s coming from the empirical distribution $p_{data}(\mathbf{x})$ are called *real* and all $\mathbf{x}$'s generated by $G_\beta(\mathbf{z})$ are dubbed *fake*. Then, we construct the objective function as follows:

- We have two sources of data: $\mathbf{x} \sim p_\theta(\mathbf{x}) = \int G_\beta(\mathbf{z})\ p(\mathbf{z})\ \mathrm{d}\mathbf{z}$ and $\mathbf{x} \sim p_{data}(\mathbf{x})$.
- The discriminator solves the classification task by assigning 0 to all fake datapoints and 1 to all real datapoints.
- Since the discriminator can be seen as a classifier, we can use the binary cross-entropy loss function in the following form:

$$\ell(\alpha, \beta) = \mathbb{E}_{\mathbf{x} \sim p_{real}} \left[\log D_\alpha(\mathbf{x})\right] + \mathbb{E}_{\mathbf{z} \sim p(\mathbf{z})} \left[\log \left(1 - D_\alpha \left(G_\beta(\mathbf{z})\right)\right)\right]. \tag{7.8}$$

  The left part corresponds to the real data source, and the right part contains the fake data source.
- We try to maximize $\ell(\alpha, \beta)$ with respect to α (i.e., the discriminator). In plain words, we want the discriminator to be as good as possible.
- The generator tries to fool the discriminator and, thus, it tries to minimize $\ell(\alpha, \beta)$ with respect to β (i.e., the generator).

  Eventually, we face the following learning objective:

$$\min_\beta \max_\alpha \mathbb{E}_{\mathbf{x} \sim p_{real}} \left[\log D_\alpha(\mathbf{x})\right] + \mathbb{E}_{\mathbf{z} \sim p(\mathbf{z})} \left[\log \left(1 - D_\alpha \left(G_\beta(\mathbf{z})\right)\right)\right]. \tag{7.9}$$

We refer to $\ell(\alpha, \beta)$ as the *adversarial loss* since there are two actors trying to achieve two opposite goals.

GANs

Let us put everything together:

- We have a generator that turns noise into fake data.
- We have a discriminator that classifies given input as either fake or real.
- We parameterize the generator and the discriminator using deep neural networks.

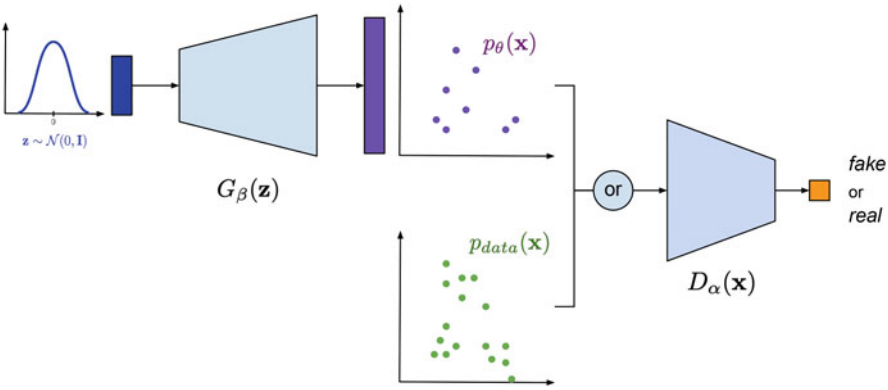

Fig. 7.2 A schematic representation of GANs. Please note the part of the generator and its resemblance to density networks

- We learn the neural networks using the adversarial loss (i.e., we optimize the min–max problem).

The resulting class of models is called Generative Adversarial Networks (GANs) [5]. In Fig. 7.2, we present the idea of GANs and how they are connected to density networks. Notice that the generator part constitutes an implicit distribution, i.e., a distribution from an unknown family of distributions, and its analytical form is unknown as well; however, we can sample from it.

7.3 Implementing GANs

Believe me or not, but we have all components to implement GANs. Let us look into all of them step-by-step. In fact, the easiest way to understand them is to implement them.

Generator

The first part is the *generator*, $G_\beta(\mathbf{z})$, which is simply a deep neural network. The code for a class of the generator is presented below. Notice that we distinguish between a function for generating, namely, transforming $\mathbf{z}$ to $\mathbf{x}$, and sampling that first samples $\mathbf{z} \sim \mathcal{N}(0, \mathbf{I})$ and then calls `generate`.

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

        # We need to init the generator neural net.
        self.generator_net = generator_net
        # We also need to know the size of the latents.
```

```
        self.z_size = z_size

    def generate(self, z):
        # Generating for given z is equivalent to applying the
    neural net.
        return self.generator_net(z)

    def sample(self, batch_size=16):
        # For sampling, we need to sample first latents.
        z = torch.randn(batch_size, self.z_size)
        return self.generate(z)

    def forward(self, z=None):
        if z is None:
            return self.sample()
        else:
            return self.generate(z)
```

Listing 7.1 A Generator class

Discriminator

The second component is the *discriminator*. Here, the code is even simpler because it consists of a single neural network. The code for a class of the discriminator is provided below:

```
class Discriminator(nn.Module):
    def __init__(self, discriminator_net):
        super(Discriminator, self).__init__()
        # We need to init the discriminator neural net.
        self.discriminator_net = discriminator_net

    def forward(self, x):
        # The forward pass is just about applying the neural net.
        return self.discriminator_net(x)
```

Listing 7.2 A Discriminator class

GAN

Now, we are ready to combine these two components. In our implementation, a GAN outputs the adversarial loss either for the generator or for the discriminator. Maybe the code below is overkill; however, it is better to write a few more lines and properly understand what is going on than applying some unclear tricks.

```
class GAN(nn.Module):
    def __init__(self, generator, discriminator, EPS=1.e-5):
        super(GAN, self).__init__()

        print('GAN by JT.')

        # To put everything together, we need the generator and
        # the discriminator. NOTE: Both are instances of classes!
        self.generator = generator
```

```
        self.discriminator = discriminator

        # For numerical issue, we introduce a small epsilon.
        self.EPS = EPS

    def forward(self, x_real, reduction='avg', mode='discriminator'):
        # The forward pass calculates the adversarial loss.
        # More specifically, either its part for the generator or
        #  the part for the discriminator.
        if mode == 'generator':
            # For the generator, we first sample FAKE data.
            x_fake_gen = self.generator.sample(x_real.shape[0])

            # Then, we calculate outputs of the discriminator for the FAKE data.
            # NOTE: We clamp here for the numerical stability later on.
            d_fake = torch.clamp(self.discriminator(x_fake_gen), self.EPS, 1. - self.EPS)

            # The loss for the generator is log(1 - D(G(z))).
            loss = torch.log(1. - d_fake)

        elif mode == 'discriminator':
            # For the discriminator, we first sample FAKE data.
            x_fake_gen = self.generator.sample(x_real.shape[0])

            # Then, we calculate outputs of the discriminator for the FAKE data.
            # NOTE: We clamp for the numerical stability later on.
            d_fake = torch.clamp(self.discriminator(x_fake_gen), self.EPS, 1. - self.EPS)

            # Moreover, we calculate outputs of the discriminator for the REAL data.
            # NOTE: We clamp for... the numerical stability (again).
            d_real = torch.clamp(self.discriminator(x_real), self.EPS, 1. - self.EPS)

            # The final loss for the discriminator is log(1 - D(G(z))) + log D(x).
            # NOTE: We take the minus sign because we MAXIMIZE the adversarial loss wrt
            # discriminator, so we MINIMIZE the negative adversarial loss wrt discriminator.
            loss = -(torch.log(d_real) + torch.log(1. - d_fake))

        if reduction == 'sum':
            return loss.sum()
        else:
            return loss.mean()
```

```
    def sample(self, batch_size=64):
        return self.generator.sample(batch_size=batch_size)
```

Listing 7.3 A GAN class

Examples of architectures for a generator and a discriminator are presented in the code below:

```
# First, we initialize the generator and the discriminator
# -generator
generator_net = nn.Sequential(nn.Linear(L, M), nn.ReLU(),
                              nn.Linear(M, D), nn.Tanh())

generator = Generator(generator_net, z_size=L)

# -discriminator
discriminator_net = nn.Sequential(nn.Linear(D, M), nn.ReLU(),
                                  nn.Linear(M, 1), nn.Sigmoid())

discriminator = Discriminator(discriminator_net)

# Eventually, we initialize the full model
model = GAN(generator=generator, discriminator=discriminator)
```

Listing 7.4 Examples of architectures

Training

One might think that the training procedure for GANs is more complicated than for any of the likelihood-based models. However, it is not the case. The only difference is that we need **two optimizers** instead of one. An example of a code with a training loop is presented below:

```
# We use two optimizers:
# optimizer_dis - an optimizer that takes the parameters of the
    discriminator
# optimizer_gen - an optimizer that takes the parameters of the
    generator
for indx_batch, batch in enumerate(training_loader):

    # -Discriminator
    # Notice that we call our model with the 'discriminator' mode
    .
    loss_dis = model.forward(batch, mode='discriminator')

    optimizer_dis.zero_grad()
    optimizer_gen.zero_grad()
    loss_dis.backward(retain_graph=True)
    optimizer_dis.step()

    # -Generator
    # Notice that we call our model with the 'generator' mode.
    loss_gen = model.forward(batch, mode='generator')
```

```
18
19      optimizer_dis.zero_grad()
20      optimizer_gen.zero_grad()
21      loss_gen.backward(retain_graph=True)
22      optimizer_gen.step()
```

Listing 7.5 A training loop

Results and Comments

In the experiments, we normalized images and scaled them to $[-1, 1]$ as we did for EBMs. The full code (with auxiliary functions) that you can play with is available here: https://github.com/jmtomczak/intro_dgm. After running it, you can expect similar results to those in Fig. 7.3.

In the previous chapters, we did not comment on the results. However, we make an exception here. Please note, my curious reader, that now we do not have a nicely converging objective. On the contrary, the adversarial loss or its generating part is jumping all over the place. That is a known fact following from the min–max optimization problem. Moreover, the loss is learnable now so it is troublesome to say where the optimal solution is since we update the loss function as well.

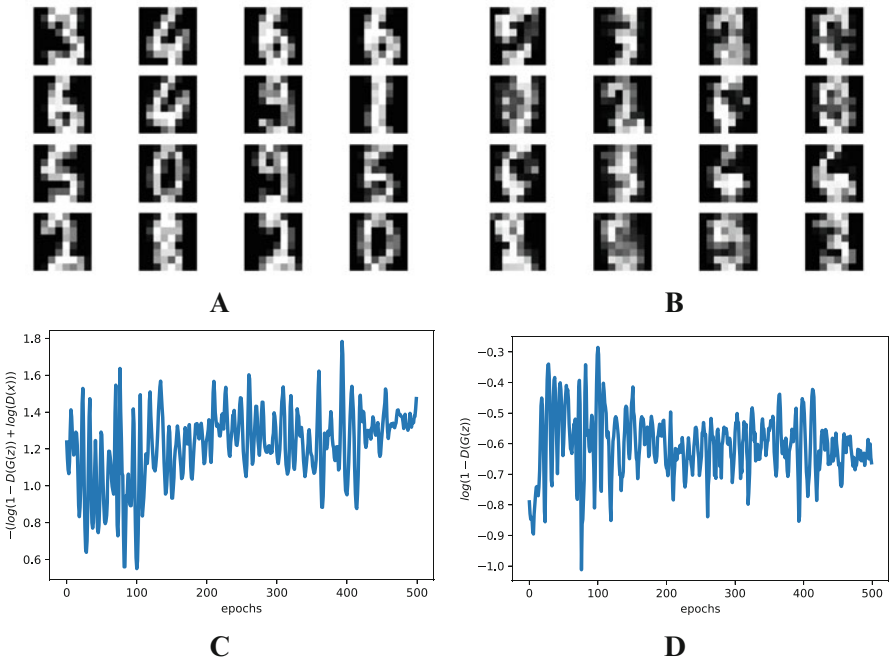

Fig. 7.3 Examples of results after running the code for GANs. (**a**) Real images. (**b**) Fake images. (**c**) The validation curve for the discriminator. (**d**) The validation curve for the generator

Fig. 7.4 Generated images after (**a**) 10 epochs of training and (**b**) 50 epochs of training

Another important piece of information is that training GANs is indeed a pain. First, it is hard to decipher and properly understand the values of the adversarial loss. Second, learning is rather slow and requires many iterations (by many I mean hundreds if not thousands). If you look into generations in the first few epochs (e.g., see Fig. 7.4), you may be discouraged because a model may seem to overfit. That is the problem, we must be really patient to see whether we are on the good track. Moreover, you may also need to pay special attention to hyperparameters, e.g., learning rates. It requires a bit of experience or simply time to play around with learning rate values in your problem.

Once you get through learning GANs, the reward is truly amazing. In the presented problem, with extremely simple neural nets, we are able to synthesize digits of high quality. That's the biggest advantage of GANs!

7.4 There Are Many GANs Out There!

Since the publication of the seminal paper on GANs [5] (however, the idea of the adversarial problem could be traced back to [6]), there was a flood of GAN-based ideas and papers. I would not even dare to mention a small fraction of them. The field of implicit modeling with GANs is growing constantly. I will try to point to a few important papers:

- *Conditional GANs*: An important extension of GANs is allowing them to generate data conditionally [7].
- *GANs with encoders*: An interesting question is whether we can extend conditional GANs to a framework with encoders. It turns out that it is possible; see BiGAN [8] and ALI [9] for details.
- *StyleGAN* and *CycleGAN*: The flexibility of GANs could be utilized in formulating specialized image synthesizers. For instance, StyleGAN is formulated in such a way to transfer style between images [10], while CycleGAN tries to "translate" one image into another, e.g., a horse into a zebra [11].

- *Wasserstein GANs*: In [12] it was claimed that the adversarial loss could be formulated differently using the Wasserstein distance (a.k.a. the earth-mover distance), that is:

$$\ell_W(\alpha, \beta) = \mathbb{E}_{\mathbf{x}\sim p_{real}}\left[D_\alpha(\mathbf{x})\right] - \mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})}\left[D_\alpha\left(G_\beta(\mathbf{z})\right)\right]. \tag{7.10}$$

  where $D_\alpha(\cdot)$ must be a 1-Lipschitz function. The simpler way to achieve that is by clipping the weight of the discriminator to some small value c. Alternatively, *spectral normalization* could be applied [13] by using the power iteration method. Overall, constraining the discriminator to be a 1-Lipshitz function stabilizes training; however, it is still hard to comprehend the learning process.
- *f-GANs*: The Wasserstein GAN indicated that we can look elsewhere for alternative formulations of the adversarial loss. In [14], it is advocated to use f-divergences for that.
- *Generative Moment Matching Networks* [15, 16]: As mentioned earlier, we could use other metrics instead of the likelihood function. We can fix the discriminator and define it as the Maximum Mean Discrepancy with a given kernel function. The resulting problem is simpler because we do not train the discriminator and, thus, we get rid of the cumbersome min–max optimization. However, the final quality of synthesized images is typically poorer.
- *Density difference vs. Density ratio*: An interesting perspective is presented in [17, 18] where we can see various GANs either as a difference of densities or as a ratio of densities. I refer to the original papers for further details.
- *Hierarchical implicit models*: The idea of defining implicit models could be extended to hierarchical models [19].
- *GANs and EBMs*: If you recall the EBMs, you may notice that there is a clear connection between the adversarial loss and the logarithm of the Boltzmann distribution. In [20, 21] it was noticed that introducing a variational distribution over observables, $q(\mathbf{x})$, leads to the following objective:

$$\mathcal{J}(\mathbf{x}) = \mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})}\left[E(\mathbf{x})\right] - \mathbb{E}_{\mathbf{x}\sim q(\mathbf{x})}\left[E(\mathbf{x})\right] + \mathbb{H}\left[q(\mathbf{x})\right], \tag{7.11}$$

  where $E(\cdot)$ is the energy function and $\mathbb{H}[\cdot]$ is the entropy. The problem again boils down to the min–max optimization problem, namely, minimizing with respect to the energy function and maximizing with respect to the variational distribution. The second difference between the adversarial loss and the variational lower bound here is the entropy term that is typically intractable.
- *What GAN to use?*: That is the question! Interestingly, it seems that training GANs greatly depends on the initialization and the neural nets rather than the adversarial loss or other tricks. You can read more about it in [22].
- *Training instabilities*: The main problem of GANs is unstable learning and a phenomenon called *mode collapse*, namely, a GAN samples beautiful images but only from some regions of the observable space. This problem has been studied for a long time by many (e.g., [23–25]); however, it still remains an open question.