the desired distribution:

$$y = \mu + \sigma z \tag{20.55}$$

We are now able to back-propagate through the sampling operation, by regarding it as a deterministic operation with an extra input z. Crucially, the extra input is a random variable whose distribution is not a function of any of the variables whose derivatives we want to calculate. The result tells us how an infinitesimal change in μ or σ would change the output if we could repeat the sampling operation again with the same value of z.

Being able to back-propagate through this sampling operation allows us to incorporate it into a larger graph. We can build elements of the graph on top of the output of the sampling distribution. For example, we can compute the derivatives of some loss function $J(y)$. We can also build elements of the graph whose outputs are the inputs or the parameters of the sampling operation. For example, we could build a larger graph with $\mu = f(\boldsymbol{x}; \boldsymbol{\theta})$ and $\sigma = g(\boldsymbol{x}; \boldsymbol{\theta})$. In this augmented graph, we can use back-propagation through these functions to derive $\nabla_{\boldsymbol{\theta}} J(y)$.

The principle used in this Gaussian sampling example is more generally applicable. We can express any probability distribution of the form $p(\mathrm{y}; \boldsymbol{\theta})$ or $p(\mathrm{y} \mid \boldsymbol{x}; \boldsymbol{\theta})$ as $p(\mathrm{y} \mid \boldsymbol{\omega})$, where $\boldsymbol{\omega}$ is a variable containing both parameters $\boldsymbol{\theta}$, and if applicable, the inputs $\boldsymbol{x}$. Given a value y sampled from distribution $p(\mathrm{y} \mid \boldsymbol{\omega})$, where $\boldsymbol{\omega}$ may in turn be a function of other variables, we can rewrite

$$\mathbf{y} \sim p(\mathbf{y} \mid \boldsymbol{\omega}) \tag{20.56}$$

as

$$\boldsymbol{y} = f(\boldsymbol{z}; \boldsymbol{\omega}), \tag{20.57}$$

where $\boldsymbol{z}$ is a source of randomness. We may then compute the derivatives of $\boldsymbol{y}$ with respect to $\boldsymbol{\omega}$ using traditional tools such as the back-propagation algorithm applied to f, so long as f is continuous and differentiable almost everywhere. Crucially, $\boldsymbol{\omega}$ must not be a function of $\boldsymbol{z}$, and $\boldsymbol{z}$ must not be a function of $\boldsymbol{\omega}$. This technique is often called the *reparametrization trick*, *stochastic back-propagation* or *perturbation analysis*.

The requirement that f be continuous and differentiable of course requires $\boldsymbol{y}$ to be continuous. If we wish to back-propagate through a sampling process that produces discrete-valued samples, it may still be possible to estimate a gradient on $\boldsymbol{\omega}$, using reinforcement learning algorithms such as variants of the REINFORCE algorithm (Williams, 1992), discussed in Sec. 20.9.1.

In neural network applications, we typically choose $\boldsymbol{z}$ to be drawn from some simple distribution, such as a unit uniform or unit Gaussian distribution, and

achieve more complex distributions by allowing the deterministic portion of the network to reshape its input.

The idea of propagating gradients or optimizing through stochastic operations dates back to the mid-twentieth century (Price, 1958; Bonnet, 1964) and was first used for machine learning in the context of reinforcement learning (Williams, 1992). More recently, it has been applied to variational approximations (Opper and Archambeau, 2009) and stochastic or generative neural networks (Bengio *et al.*, 2013b; Kingma, 2013; Kingma and Welling, 2014b,a; Rezende *et al.*, 2014; Goodfellow *et al.*, 2014c). Many networks, such as denoising autoencoders or networks regularized with dropout, are also naturally designed to take noise as an input without requiring any special reparametrization to make the noise independent from the model.

20.9.1 Back-Propagating through Discrete Stochastic Operations

When a model emits a discrete variable $\boldsymbol{y}$, the reparametrization trick is not applicable. Suppose that the model takes inputs $\boldsymbol{x}$ and parameters $\boldsymbol{\theta}$, both encapsulated in the vector $\boldsymbol{\omega}$, and combines them with random noise $\boldsymbol{z}$ to produce $\boldsymbol{y}$:

$$\boldsymbol{y} = f(\boldsymbol{z}; \boldsymbol{\omega}). \tag{20.58}$$

Because $\boldsymbol{y}$ is discrete, f must be a step function. The derivatives of a step function are not useful at any point. Right at each step boundary, the derivatives are undefined, but that is a small problem. The large problem is that the derivatives are zero almost everywhere, on the regions between step boundaries. The derivatives of any cost function $J(\boldsymbol{y})$ therefore do not give any information for how to update the model parameters $\boldsymbol{\theta}$.

The REINFORCE algorithm (REward Increment = Non-negative Factor $\times$ Offset Reinforcement $\times$ Characteristic Eligibility) provides a framework defining a family of simple but powerful solutions (Williams, 1992). The core idea is that even though $J(f(\boldsymbol{z}; \boldsymbol{\omega}))$ is a step function with useless derivatives, the expected cost $\mathbb{E}_{\mathbf{z} \sim p(\mathbf{z})} J\left(f(\boldsymbol{z}; \boldsymbol{\omega})\right)$ is often a smooth function amenable to gradient descent. Although that expectation is typically not tractable when $\boldsymbol{y}$ is high-dimensional (or is the result of the composition of many discrete stochastic decisions), it can be estimated without bias using a Monte Carlo average. The stochastic estimate of the gradient can be used with SGD or other stochastic gradient-based optimization techniques.

The simplest version of REINFORCE can be derived by simply differentiating

the expected cost:

$$\mathbb{E}_z[J(\boldsymbol{y})] = \sum_{\boldsymbol{y}} J(\boldsymbol{y}) p(\boldsymbol{y}) \tag{20.59}$$

$$\frac{\partial \mathbb{E}[J(\boldsymbol{y})]}{\partial \boldsymbol{\omega}} = \sum_{y} J(\boldsymbol{y}) \frac{\partial p(\boldsymbol{y})}{\partial \boldsymbol{\omega}} \tag{20.60}$$

$$= \sum_{y} J(\boldsymbol{y}) p(\boldsymbol{y}) \frac{\partial \log p(\boldsymbol{y})}{\partial \boldsymbol{\omega}} \tag{20.61}$$

$$\approx \frac{1}{m} \sum_{\mathbf{y}^{(i)} \sim p(\boldsymbol{y}),\, i=1}^{m} J(\boldsymbol{y}^{(i)}) \frac{\partial \log p(\boldsymbol{y}^{(i)})}{\partial \boldsymbol{\omega}}. \tag{20.62}$$

Eq. 20.60 relies on the assumption that J does not reference ω directly. It is trivial to extend the approach to relax this assumption. Eq. 20.61 exploits the derivative rule for the logarithm, $\frac{\partial \log p(\boldsymbol{y})}{\partial \boldsymbol{\omega}} = \frac{1}{p(\boldsymbol{y})} \frac{\partial p(\boldsymbol{y})}{\partial \boldsymbol{\omega}}$. Eq. 20.62 gives an unbiased Monte Carlo estimator of the gradient.

Anywhere we write $p(\boldsymbol{y})$ in this section, one could equally write $p(\boldsymbol{y} \mid \boldsymbol{x})$. This is because $p(\boldsymbol{y})$ is parametrized by $\boldsymbol{\omega}$, and $\boldsymbol{\omega}$ contains both $\boldsymbol{\theta}$ and $\boldsymbol{x}$, if $\boldsymbol{x}$ is present.

One issue with the above simple REINFORCE estimator is that it has a very high variance, so that many samples of $\boldsymbol{y}$ need to be drawn to obtain a good estimator of the gradient, or equivalently, if only one sample is drawn, SGD will converge very slowly and will require a smaller learning rate. It is possible to considerably reduce the variance of that estimator by using *variance reduction* methods (Wilson, 1984; L'Ecuyer, 1994). The idea is to modify the estimator so that its expected value remains unchanged but its variance get reduced. In the context of REINFORCE, the proposed variance reduction methods involve the computation of a *baseline* that is used to offset $J(\boldsymbol{y})$. Note that any offset $b(\boldsymbol{\omega})$ that does not depend on $\boldsymbol{y}$ would not change the expectation of the estimated gradient because

$$E_{p(\boldsymbol{y})}\left[\frac{\partial \log p(\boldsymbol{y})}{\partial \boldsymbol{\omega}}\right] = \sum_{\boldsymbol{y}} p(\boldsymbol{y}) \frac{\partial \log p(\boldsymbol{y})}{\partial \boldsymbol{\omega}} \tag{20.63}$$

$$= \sum_{\boldsymbol{y}} \frac{\partial p(\boldsymbol{y})}{\partial \boldsymbol{\omega}} \tag{20.64}$$

$$= \frac{\partial}{\partial \boldsymbol{\omega}} \sum_{\boldsymbol{y}} p(\boldsymbol{y}) = \frac{\partial}{\partial \boldsymbol{\omega}} 1 = 0, \tag{20.65}$$

which means that

$$E_{p(\boldsymbol{y})}\left[(J(\boldsymbol{y})-b(\boldsymbol{\omega}))\frac{\partial \log p(\boldsymbol{y})}{\partial \boldsymbol{\omega}}\right] = E_{p(\boldsymbol{y})}\left[J(\boldsymbol{y})\frac{\partial \log p(\boldsymbol{y})}{\partial \boldsymbol{\omega}}\right] - b(\boldsymbol{\omega})E_{p(\boldsymbol{y})}\left[\frac{\partial \log p(\boldsymbol{y})}{\partial \boldsymbol{\omega}}\right] \tag{20.66}$$

$$= E_{p(\boldsymbol{y})}\left[J(\boldsymbol{y})\frac{\partial \log p(\boldsymbol{y})}{\partial \boldsymbol{\omega}}\right]. \tag{20.67}$$

Furthermore, we can obtain the optimal $b(\boldsymbol{\omega})$ by computing the variance of $(J(\boldsymbol{y}) - b(\boldsymbol{\omega}))\frac{\partial \log p(\boldsymbol{y})}{\partial \boldsymbol{\omega}}$ under $p(\boldsymbol{y})$ and minimizing with respect to $b(\boldsymbol{\omega})$. What we find is that this optimal baseline $b^*(\boldsymbol{\omega})_i$ is different for each element ω_i of the vector $\boldsymbol{\omega}$:

$$b^*(\boldsymbol{\omega})_i = \frac{E_{p(\boldsymbol{y})}\left[J(\boldsymbol{y})\frac{\partial \log p(\boldsymbol{y})}{\partial \omega_i}^2\right]}{E_{p(\boldsymbol{y})}\left[\frac{\partial \log p(\boldsymbol{y})}{\partial \omega_i}^2\right]}. \tag{20.68}$$

The gradient estimator with respect to ω_i then becomes

$$(J(\boldsymbol{y}) - b(\boldsymbol{\omega})_i)\frac{\partial \log p(\boldsymbol{y})}{\partial \omega_i} \tag{20.69}$$

where $b(\boldsymbol{\omega})_i$ estimates the above $b^*(\boldsymbol{\omega})_i$. The estimate b is usually obtained by adding extra outputs to the neural network and training the new outputs to estimate $E_{p(\boldsymbol{y})}\left[J(\boldsymbol{y})\frac{\partial \log p(\boldsymbol{y})}{\partial \omega_i}^2\right]$ and $E_{p(\boldsymbol{y})}\left[\frac{\partial \log p(\boldsymbol{y})}{\partial \omega_i}^2\right]$ for each element of $\boldsymbol{\omega}$. These extra outputs can be trained with the mean squared error objective, using respectively $J(\boldsymbol{y})\frac{\partial \log p(\boldsymbol{y})}{\partial \omega_i}^2$ and $\frac{\partial \log p(\boldsymbol{y})}{\partial \omega_i}^2$ as targets when $\boldsymbol{y}$ is sampled from $p(\boldsymbol{y})$, for a given $\boldsymbol{\omega}$. The estimate b may then be recovered by substituting these estimates into Eq. 20.68. Mnih and Gregor (2014) preferred to use a single shared output (across all elements i of $\boldsymbol{\omega}$) trained with the target $J(\boldsymbol{y})$, using as baseline $b(\boldsymbol{\omega}) \approx E_{p(\boldsymbol{y})}[J(\boldsymbol{y})]$.

Variance reduction methods have been introduced in the reinforcement learning context (Sutton *et al.*, 2000; Weaver and Tao, 2001), generalizing previous work on the case of binary reward by Dayan (1990). See Bengio *et al.* (2013b), Mnih and Gregor (2014), Ba *et al.* (2014), Mnih *et al.* (2014), or Xu *et al.* (2015) for examples of modern uses of the REINFORCE algorithm with reduced variance in the context of deep learning. In addition to the use of an input-dependent baseline $b(\boldsymbol{\omega})$, Mnih and Gregor (2014) found that the scale of $(J(\boldsymbol{y}) - b(\boldsymbol{\omega}))$ could be adjusted during training by dividing it by its standard deviation estimated by a moving average during training, as a kind of adaptive learning rate, to counter the effect of important variations that occur during the course of training in the magnitude of this quantity. Mnih and Gregor (2014) called this heuristic *variance normalization*.

REINFORCE-based estimators can be understood as estimating the gradient by correlating choices of $\boldsymbol{y}$ with corresponding values of $J(\boldsymbol{y})$. If a good value of $\boldsymbol{y}$ is unlikely under the current parametrization, it might take a long time to obtain it by chance, and get the required signal that this configuration should be reinforced.

20.10 Directed Generative Nets

As discussed in Chapter 16, directed graphical models make up a prominent class of graphical models. While directed graphical models have been very popular within the greater machine learning community, within the smaller deep learning community they have until roughly 2013 been overshadowed by undirected models such as the RBM.

In this section we review some of the standard directed graphical models that have traditionally been associated with the deep learning community.

We have already described deep belief networks, which are a partially directed model. We have also already described sparse coding models, which can be thought of as shallow directed generative models. They are often used as feature learners in the context of deep learning, though they tend to perform poorly at sample generation and density estimation. We now describe a variety of deep, fully directed models.

20.10.1 Sigmoid Belief Nets

Sigmoid belief networks (Neal, 1990) are a simple form of directed graphical model with a specific kind of conditional probability distribution. In general, we can think of a sigmoid belief network as having a vector of binary states $\boldsymbol{s}$, with each element of the state influenced by its ancestors:

$$p(s_i) = \sigma\left(\sum_{j<i} W_{j,i} s_j + b_i\right). \tag{20.70}$$

The most common structure of sigmoid belief network is one that is divided into many layers, with ancestral sampling proceeding through a series of many hidden layers and then ultimately generating the visible layer. This structure is very similar to the deep belief network, except that the units at the beginning of the sampling process are independent from each other, rather than sampled from a restricted Boltzmann machine. Such a structure is interesting for a variety of

reasons. One reason is that the structure is a universal approximator of probability distributions over the visible units, in the sense that it can approximate any probability distribution over binary variables arbitrarily well, given enough depth, even if the width of the individual layers is restricted to the dimensionality of the visible layer (Sutskever and Hinton, 2008).

While generating a sample of the visible units is very efficient in a sigmoid belief network, most other operations are not. Inference over the hidden units given the visible units is intractable. Mean field inference is also intractable because the variational lower bound involves taking expectations of cliques that encompass entire layers. This problem has remained difficult enough to restrict the popularity of directed discrete networks.

One approach for performing inference in a sigmoid belief network is to construct a different lower bound that is specialized for sigmoid belief networks (Saul *et al.*, 1996). This approach has only been applied to very small networks. Another approach is to use learned inference mechanisms as described in Sec. 19.5. The Helmholtz machine (Dayan *et al.*, 1995; Dayan and Hinton, 1996) is a sigmoid belief network combined with an inference network that predicts the parameters of the mean field distribution over the hidden units. Modern approaches (Gregor *et al.*, 2014; Mnih and Gregor, 2014) to sigmoid belief networks still use this inference network approach. These techniques remain difficult due to the discrete nature of the latent variables. One cannot simply back-propagate through the output of the inference network, but instead must use the relatively unreliable machinery for back-propagating through discrete sampling processes, described in Sec. 20.9.1. Recent approaches based on importance sampling, reweighted wake-sleep (Bornschein and Bengio, 2015) and bidirectional Helmholtz machines (Bornschein *et al.*, 2015) make it possible to quickly train sigmoid belief networks and reach state-of-the-art performance on benchmark tasks.

A special case of sigmoid belief networks is the case where there are no latent variables. Learning in this case is efficient, because there is no need to marginalize latent variables out of the likelihood. A family of models called auto-regressive networks generalize this fully visible belief network to other kinds of variables besides binary variables and other structures of conditional distributions besides log-linear relationships. Auto-regressive networks are described later, in Sec. 20.10.7.

20.10.2 Differentiable Generator Nets

Many generative models are based on the idea of using a differentiable *generator network*. The model transforms samples of latent variables $\mathbf{z}$ to samples $\mathbf{x}$ or

to distributions over samples $\mathbf{x}$ using a differentiable function $g(\boldsymbol{z}; \boldsymbol{\theta}^{(g)})$ which is typically represented by a neural network. This model class includes variational autoencoders, which pair the generator net with an inference net, generative adversarial networks, which pair the generator network with a discriminator network, and techniques that train generator networks in isolation.

Generator networks are essentially just parametrized computational procedures for generating samples, where the architecture provides the family of possible distributions to sample from and the parameters select a distribution from within that family.

As an example, the standard procedure for drawing samples from a normal distribution with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$ is to feed samples $\boldsymbol{z}$ from a normal distribution with zero mean and identity covariance into a very simple generator network. This generator network contains just one affine layer:

$$\boldsymbol{x} = g(\boldsymbol{z}) = \mu + \boldsymbol{L}\boldsymbol{z} \tag{20.71}$$

where $\boldsymbol{L}$ is given by the Cholesky decomposition of $\boldsymbol{\Sigma}$.

Pseudorandom number generators can also use nonlinear transformations of simple distributions. For example, *inverse transform sampling* (Devroye, 2013) draws a scalar z from $U(0, 1)$ and applies a nonlinear transformation to a scalar x. In this case $g(z)$ is given by the inverse of the cumulative distribution function $F(x) = \int_{-\infty}^{x} p(v)dv$. If we are able to specify $p(x)$, integrate over x, and invert the resulting function, we can sample from $p(x)$ without using machine learning.

To generate samples from more complicated distributions that are difficult to specify directly, difficult to integrate over, or whose resulting integrals are difficult to invert, we use a feedforward network to represent a parametric family of nonlinear functions g, and use training data to infer the parameters selecting the desired function.

We can think of g as providing a nonlinear change of variables that transforms the distribution over $\mathbf{z}$ into the desired distribution over $\mathbf{x}$.

Recall from Eq. 3.47 that, for invertible, differentiable, continuous g,

$$p_z(\boldsymbol{z}) = p_x(g(\boldsymbol{z})) \left| \det\left(\frac{\partial g}{\partial \boldsymbol{z}}\right) \right|. \tag{20.72}$$

This implicitly imposes a probability distribution over $\mathbf{x}$:

$$p_x(\boldsymbol{x}) = \frac{p_z(g^{-1}(\boldsymbol{x}))}{\left| \det\left(\frac{\partial g}{\partial \boldsymbol{z}}\right) \right|}. \tag{20.73}$$

Of course, this formula may be difficult to evaluate, depending on the choice of g, so we often use indirect means of learning g, rather than trying to maximize $\log p(\boldsymbol{x})$ directly.

In some cases, rather than using g to provide a sample of $\boldsymbol{x}$ directly, we use g to define a conditional distribution over $\boldsymbol{x}$. For example, we could use a generator net whose final layer consists of sigmoid outputs to provide the mean parameters of Bernoulli distributions:

$$p(\mathrm{x}_i = 1 \mid \boldsymbol{z}) = g(\boldsymbol{z})_i. \tag{20.74}$$

In this case, when we use g to define $p(\boldsymbol{x} \mid \boldsymbol{z})$, we impose a distribution over $\boldsymbol{x}$ by marginalizing $\boldsymbol{z}$:

$$p(\boldsymbol{x}) = \mathbb{E}_{\boldsymbol{z}} p(\boldsymbol{x} \mid \boldsymbol{z}). \tag{20.75}$$

Both approaches define a distribution $p_g(\boldsymbol{x})$ and allow us to train various criteria of p_g using the reparametrization trick of Sec. 20.9.

The two different approaches to formulating generator nets—emitting the parameters of a conditional distribution versus directly emitting samples—have complementary strengths and weaknesses. When the generator net defines a conditional distribution over $\boldsymbol{x}$, it is capable of generating discrete data as well as continuous data. When the generator net provides samples directly, it is capable of generating only continuous data (we could introduce discretization in the forward propagation, but this would lose the ability to learn the model using back-propagation). The advantage to direct sampling is that we are no longer forced to use conditional distributions whose form can be easily written down and algebraically manipulated by a human designer.

Approaches based on differentiable generator networks are motivated by the success of gradient descent applied to differentiable feedforward networks for classification. In the context of supervised learning, deep feedforward networks trained with gradient-based learning seem practically guaranteed to succeed given enough hidden units and enough training data. Can this same recipe for success transfer to generative modeling?

Generative modeling seems to be more difficult than classification or regression because the learning process requires optimizing intractable criteria. In the context of differentiable generator nets, the criteria are intractable because the data does not specify both the inputs $\boldsymbol{z}$ and the outputs $\boldsymbol{x}$ of the generator net. In the case of supervised learning, both the inputs $\boldsymbol{x}$ and the outputs $\boldsymbol{y}$ were given, and the optimization procedure needs only to learn how to produce the specified mapping. In the case of generative modeling, the learning procedure needs to determine how to arrange $\boldsymbol{z}$ space in a useful way and additionally how to map from $\boldsymbol{z}$ to $\boldsymbol{x}$.

Dosovitskiy *et al.* (2015) studied a simplified problem, where the correspondence between $\boldsymbol{z}$ and $\boldsymbol{x}$ is given. Specifically, the training data is computer-rendered imagery of chairs. The latent variables $\boldsymbol{z}$ are parameters given to the rendering engine describing the choice of which chair model to use, the position of the chair, and other configuration details that affect the rendering of the image. Using this synthetically generated data, a convolutional network is able to learn to map $\boldsymbol{z}$ descriptions of the content of an image to $\boldsymbol{x}$ approximations of rendered images. This suggests that contemporary differentiable generator networks have sufficient model capacity to be good generative models, and that contemporary optimization algorithms have the ability to fit them. The difficulty lies in determining how to train generator networks when the value of $\boldsymbol{z}$ for each $\boldsymbol{x}$ is not fixed and known ahead of each time.

The following sections describe several approaches to training differentiable generator nets given only training samples of $\boldsymbol{x}$.

20.10.3 Variational Autoencoders

The *variational autoencoder* or *VAE* (Kingma, 2013; Rezende *et al.*, 2014) is a directed model that uses learned approximate inference and can be trained purely with gradient-based methods.

To generate a sample from the model, the VAE first draws a sample $\boldsymbol{z}$ from the code distribution $p_{\text{model}}(\boldsymbol{z})$. The sample is then run through a differentiable generator network $g(\boldsymbol{z})$. Finally, $\boldsymbol{x}$ is sampled from a distribution $p_{\text{model}}(\boldsymbol{x}; g(\boldsymbol{z})) = p_{\text{model}}(\boldsymbol{x} \mid \boldsymbol{z})$. However, during training, the approximate inference network (or encoder) $q(\boldsymbol{z} \mid \boldsymbol{x})$ is used to obtain $\boldsymbol{z}$ and $p_{\text{model}}(\boldsymbol{x} \mid \boldsymbol{z})$ is then viewed as a decoder network.

The key insight behind variational autoencoders is that they may be trained by maximizing the variational lower bound $\mathcal{L}(q)$ associated with data point $\boldsymbol{x}$:

$$\mathcal{L}(q) = \mathbb{E}_{\boldsymbol{z} \sim q(\boldsymbol{z}|\boldsymbol{x})} \log p_{\text{model}}(\boldsymbol{z}, \boldsymbol{x}) + \mathcal{H}(q(\mathbf{z} \mid \boldsymbol{x})) \tag{20.76}$$

$$= \mathbb{E}_{\boldsymbol{z} \sim q(\boldsymbol{z}|\boldsymbol{x})} \log p_{\text{model}}(\boldsymbol{x} \mid \boldsymbol{z}) - D_{\text{KL}}(q(\mathbf{z} \mid \boldsymbol{x}) || p_{\text{model}}(\mathbf{z})) \tag{20.77}$$

$$\leq \log p_{\text{model}}(\boldsymbol{x}). \tag{20.78}$$

In Eq. 20.76, we recognize the first term as the joint log-likelihood of the visible and hidden variables under the approximate posterior over the latent variables (just like with EM, except that we use an approximate rather than the exact posterior). We recognize also a second term, the entropy of the approximate posterior. When q is chosen to be a Gaussian distribution, with noise added to a predicted mean value, maximizing this entropy term encourages increasing the standard deviation

of this noise. More generally, this entropy term encourages the variational posterior to place high probability mass on many $\boldsymbol{z}$ values that could have generated $\boldsymbol{x}$, rather than collapsing to a single point estimate of the most likely value. In Eq. 20.77, we recognize the first term as the reconstruction log-likelihood found in other autoencoders. The second term tries to make the approximate posterior distribution $q(\mathbf{z} \mid \boldsymbol{x})$ and the model prior $p_{\text{model}}(\boldsymbol{z})$ approach each other.

Traditional approaches to variational inference and learning infer q via an optimization algorithm, typically iterated fixed point equations (Sec. 19.4). These approaches are slow and often require the ability to compute $\mathbb{E}_{\mathrm{z}\sim q} \log p_{\text{model}}(\boldsymbol{z}, \boldsymbol{x})$ in closed form. The main idea behind the variational autoencoder is to train a parametric encoder (also sometimes called an inference network or recognition model) that produces the parameters of q. So long as $\boldsymbol{z}$ is a continuous variable, we can then back-propagate through samples of $\boldsymbol{z}$ drawn from $q(\boldsymbol{z} \mid \boldsymbol{x}) = q(\boldsymbol{z}; f(\boldsymbol{x}; \boldsymbol{\theta}))$ in order to obtain a gradient with respect to $\boldsymbol{\theta}$. Learning then consists solely of maximizing $\mathcal{L}$ with respect to the parameters of the encoder and decoder. All of the expectations in $\mathcal{L}$ may be approximated by Monte Carlo sampling.

The variational autoencoder approach is elegant, theoretically pleasing, and simple to implement. It also obtains excellent results and is among the state of the art approaches to generative modeling. Its main drawback is that samples from variational autoencoders trained on images tend to be somewhat blurry. The causes of this phenomenon are not yet known. One possibility is that the blurriness is an intrinsic effect of maximum likelihood, which minimizes $D_{\text{KL}}(p_{\text{data}} \| p_{\text{model}})$. As illustrated in Fig. 3.6, this means that the model will assign high probability to points that occur in the training set, but may also assign high probability to other points. These other points may include blurry images. Part of the reason that the model would choose to put probability mass on blurry images rather than some other part of the space is that the variational autoencoders used in practice usually have a Gaussian distribution for $p_{\text{model}}(\boldsymbol{x}; g(\boldsymbol{z}))$. Maximizing a lower bound on the likelihood of such a distribution is similar to training a traditional autoencoder with mean squared error, in the sense that it has a tendency to ignore features of the input that occupy few pixels or that cause only a small change in the brightness of the pixels that they occupy. This issue is not specific to VAEs and is shared with generative models that optimize a log-likelihood, or equivalently, $D_{\text{KL}}(p_{\text{data}} \| p_{\text{model}})$, as argued by Theis *et al.* (2015) and by Huszar (2015). Another troubling issue with contemporary VAE models is that they tend to use only a small subset of the dimensions of $\boldsymbol{z}$, as if the encoder was not able to transform enough of the local directions in input space to a space where the marginal distribution matches the factorized prior.

The VAE framework is very straightforward to extend to a wide range of model architectures. This is a key advantage over Boltzmann machines, which require extremely careful model design to maintain tractability. VAEs work very well with a diverse family of differentiable operators. One particularly sophisticated VAE is the *deep recurrent attention writer* or *DRAW* model (Gregor *et al.*, 2015). DRAW uses a recurrent encoder and recurrent decoder combined with an attention mechanism. The generation process for the DRAW model consists of sequentially visiting different small image patches and drawing the values of the pixels at those points. VAEs can also be extended to generate sequences by defining variational RNNs (Chung *et al.*, 2015b) by using a recurrent encoder and decoder within the VAE framework. Generating a sample from a traditional RNN involves only non-deterministic operations at the output space. Variational RNNs also have random variability at the potentially more abstract level captured by the VAE latent variables.

The VAE framework has been extended to maximize not just the traditional variational lower bound, but instead the *importance weighted autoencoder* (Burda *et al.*, 2015) objective:

$$\mathcal{L}_k(\boldsymbol{x}, q) = \mathbb{E}_{\mathbf{z}^{(1)},\ldots,\mathbf{z}^{(k)} \sim q(\boldsymbol{z}|\boldsymbol{x})} \left[\log \frac{1}{k} \sum_{i=1}^{k} \frac{p_{\text{model}}(\boldsymbol{x}, \boldsymbol{z}^{(i)})}{q(\boldsymbol{z}^{(i)} \mid \boldsymbol{x})} \right]. \tag{20.79}$$

This new objective is equivalent to the traditional lower bound $\mathcal{L}$ when $k = 1$. However, it may also be interpreted as forming an estimate of the true $\log p_{\text{model}}(\boldsymbol{x})$ using importance sampling of $\boldsymbol{z}$ from proposal distribution $q(\boldsymbol{z} \mid \boldsymbol{x})$. The importance weighted autoencoder objective is also a lower bound on $\log p_{\text{model}}(\boldsymbol{x})$ and becomes tighter as k increases.

Variational autoencoders have some interesting connections to the MP-DBM and other approaches that involve back-propagation through the approximate inference graph (Goodfellow *et al.*, 2013b; Stoyanov *et al.*, 2011; Brakel *et al.*, 2013). These previous approaches required an inference procedure such as mean field fixed point equations to provide the computational graph. The variational autoencoder is defined for arbitrary computational graphs, which makes it applicable to a wider range of probabilistic model families because there is no need to restrict the choice of models to those with tractable mean field fixed point equations. The variational autoencoder also has the advantage that it increases a bound on the log-likelihood of the model, while the criteria for the MP-DBM and related models are more heuristic and have little probabilistic interpretation beyond making the results of approximate inference accurate. One disadvantage of the variational autoencoder is that it learns an inference network for only one problem, inferring $\boldsymbol{z}$ given $\boldsymbol{x}$.

The older methods are able to perform approximate inference over any subset of variables given any other subset of variables, because the mean field fixed point equations specify how to share parameters between the computational graphs for all of these different problems.

One very nice property of the variational autoencoder is that simultaneously training a parametric encoder in combination with the generator network forces the model to learn a predictable coordinate system that the encoder can capture. This makes it an excellent manifold learning algorithm. See Fig. 20.6 for examples of low-dimensional manifolds learned by the variational autoencoder. In one of the cases demonstrated in the figure, the algorithm discovered two independent factors of variation present in images of faces: angle of rotation and emotional expression.

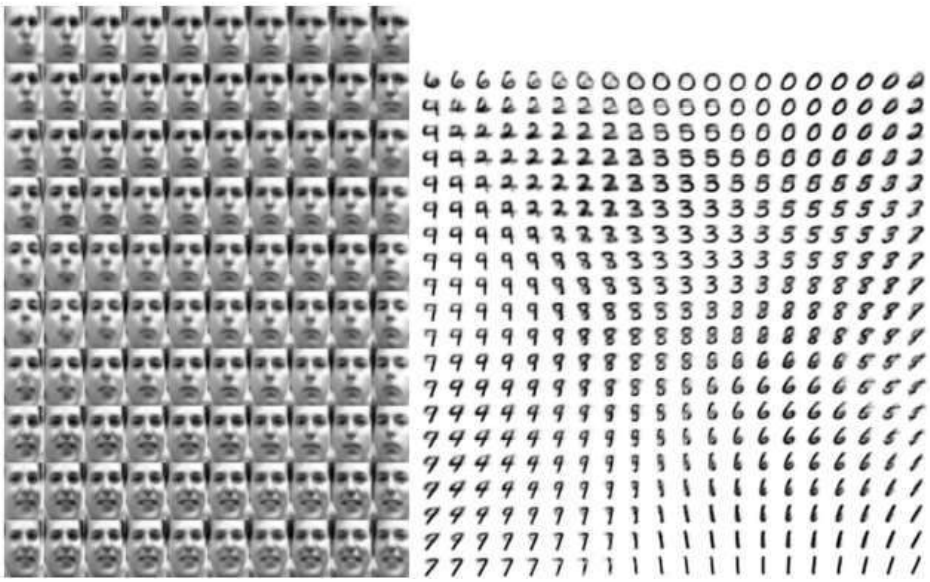

Figure 20.6: Examples of two-dimensional coordinate systems for high-dimensional manifolds, learned by a variational autoencoder (Kingma and Welling, 2014a). Two dimensions may be plotted directly on the page for visualization, so we can gain an understanding of how the model works by training a model with a 2-D latent code, even if we believe the intrinsic dimensionality of the data manifold is much higher. The images shown are not examples from the training set but images $\boldsymbol{x}$ actually generated by the model $p(\boldsymbol{x} \mid \boldsymbol{z})$, simply by changing the 2-D "code" $\boldsymbol{z}$ (each image corresponds to a different choice of "code" $\boldsymbol{z}$ on a 2-D uniform grid). *(Left)* The two-dimensional map of the Frey faces manifold. One dimension that has been discovered (horizontal) mostly corresponds to a rotation of the face, while the other (vertical) corresponds to the emotional expression. *(Right)* The two-dimensional map of the MNIST manifold.

20.10.4 Generative Adversarial Networks

Generative adversarial networks or GANs (Goodfellow *et al.*, 2014c) are another generative modeling approach based on differentiable generator networks.

Generative adversarial networks are based on a game theoretic scenario in which the generator network must compete against an adversary. The generator network directly produces samples $\boldsymbol{x} = g(\boldsymbol{z}; \boldsymbol{\theta}^{(g)})$. Its adversary, the *discriminator network*, attempts to distinguish between samples drawn from the training data and samples drawn from the generator. The discriminator emits a probability value given by $d(\boldsymbol{x}; \boldsymbol{\theta}^{(d)})$, indicating the probability that $\boldsymbol{x}$ is a real training example rather than a fake sample drawn from the model.

The simplest way to formulate learning in generative adversarial networks is as a zero-sum game, in which a function $v(\boldsymbol{\theta}^{(g)}, \boldsymbol{\theta}^{(d)})$ determines the payoff of the discriminator. The generator receives $-v(\boldsymbol{\theta}^{(g)}, \boldsymbol{\theta}^{(d)})$ as its own payoff. During learning, each player attempts to maximize its own payoff, so that at convergence

$$g^* = \arg\min_g \max_d v(g, d). \tag{20.80}$$

The default choice for v is

$$v(\boldsymbol{\theta}^{(g)}, \boldsymbol{\theta}^{(d)}) = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}} \log d(\boldsymbol{x}) + \mathbb{E}_{\boldsymbol{x} \sim p_{\text{model}}} \log \left(1 - d(\boldsymbol{x})\right). \tag{20.81}$$

This drives the discriminator to attempt to learn to correctly classify samples as real or fake. Simultaneously, the generator attempts to fool the classifier into believing its samples are real. At convergence, the generator's samples are indistinguishable from real data, and the discriminator outputs $\frac{1}{2}$ everywhere. The discriminator may then be discarded.

The main motivation for the design of GANs is that the learning process requires neither approximate inference nor approximation of a partition function gradient. In the case where $\max_d v(g, d)$ is convex in $\boldsymbol{\theta}^{(g)}$ (such as the case where optimization is performed directly in the space of probability density functions) then the procedure is guaranteed to converge and is asymptotically consistent.

Unfortunately, learning in GANs can be difficult in practice when g and d are represented by neural networks and $\max_d v(g, d)$ is not convex. Goodfellow (2014) identified non-convergence as an issue that may cause GANs to underfit. In general, simultaneous gradient descent on two players' costs is not guaranteed to reach an equilibrium. Consider for example the value function $v(a, b) = ab$, where one player controls a and incurs cost ab, while the other player controls b and receives a cost $-ab$. If we model each player as making infinitesimally small

gradient steps, each player reducing their own cost at the expense of the other player, then a and b go into a stable, circular orbit, rather than arriving at the equilibrium point at the origin. Note that the equilibria for a minimax game are not local minima of v. Instead, they are points that are simultaneously minima for both players' costs. This means that they are saddle points of v that are local minima with respect to the first player's parameters and local maxima with respect to the second player's parameters. It is possible for the two players to take turns increasing then decreasing v forever, rather than landing exactly on the saddle point where neither player is capable of reducing their cost. It is not known to what extent this non-convergence problem affects GANs.

Goodfellow (2014) identified an alternative formulation of the payoffs, in which the game is no longer zero-sum, that has the same expected gradient as maximum likelihood learning whenever the discriminator is optimal. Because maximum likelihood training converges, this reformulation of the GAN game should also converge, given enough samples. Unfortunatley, this alternative formulation does not seem to perform well in practice, possibly due to suboptimality of the discriminator, or possibly due to high variance around the expected gradient.

In practice, the best-performing formulation of the GAN game is a different formulation that is neither zero-sum nor equivalent to maximum likelihood, introduced by Goodfellow *et al.* (2014c) with a heuristic motivation. In this best-performing formulation, the generator aims to increase the log probability that the discriminator makes a mistake, rather than aiming to decrease the log probability that the discriminator makes the correct prediction. This reformulation is motivated solely by the observation that it causes the derivative of the generator's cost function with respect to the discriminator's logits to remain large even in the situation where the discriminator confidently rejects all generator samples.

Stabilization of GAN learning remains an open problem. Fortunately, GAN learning performs well when the model architecture and hyperparameters are carefully selected. Radford *et al.* (2015) crafted a deep convolutional GAN (DCGAN) that performs very well for image synthesis tasks, and showed that its latent representation space captures important factors of variation, as shown in Fig. 15.9. See Fig. 20.7 for examples of images generated by a DCGAN generator.

The GAN learning problem can also be simplified by breaking the generation process into many levels of detail. It is possible to train conditional GANs (Mirza and Osindero, 2014) that learn to sample from a distribution $p(\boldsymbol{x} \mid \boldsymbol{y})$ rather than simply sampling from a marginal distribution $p(\boldsymbol{x})$. Denton *et al.* (2015) showed that a series of conditional GANs can be trained to first generate a very low-resolution version of an image, then incrementally add details to the image.

Figure 20.7: Images generated by GANs trained on the LSUN dataset. *(Left)* Images of bedrooms generated by a DCGAN model, reproduced with permission from Radford *et al.* (2015). *(Right)* Images of churches generated by a LAPGAN model, reproduced with permission from Denton *et al.* (2015).

This technique is called the LAPGAN model, due to the use of a Laplacian pyramid to generate the images containing varying levels of detail. LAPGAN generators are able to fool not only discriminator networks but also human observers, with experimental subjects identifying up to 40% of the outputs of the network as being real data. See Fig. 20.7 for examples of images generated by a LAPGAN generator.

One unusual capability of the GAN training procedure is that it can fit probability distributions that assign zero probability to the training points. Rather than maximizing the log probability of specific points, the generator net learns to trace out a manifold whose points resemble training points in some way. Somewhat paradoxically, this means that the model may assign a log-likelihood of negative infinity to the test set, while still representing a manifold that a human observer judges to capture the essence of the generation task. This is not clearly an advantage or a disadvantage, and one may also guarantee that the generator network assigns non-zero probability to all points simply by making the last layer of the generator network add Gaussian noise to all of the generated values. Generator networks that add Gaussian noise in this manner sample from the same distribution that one obtains by using the generator network to parametrize the mean of a conditional Gaussian distribution.

Dropout seems to be important in the discriminator network. In particular, units should be stochastically dropped while computing the gradient for the generator network to follow. Following the gradient of the deterministic version of the discriminator with its weights divided by two does not seem to be as effective.

Likewise, never using dropout seems to yield poor results.

While the GAN framework is designed for differentiable generator networks, similar principles can be used to train other kinds of models. For example, *self-supervised boosting* can be used to train an RBM generator to fool a logistic regression discriminator (Welling *et al.*, 2002).

20.10.5 Generative Moment Matching Networks

Generative moment matching networks (Li *et al.*, 2015; Dziugaite *et al.*, 2015) are another form of generative model based on differentiable generator networks. Unlike VAEs and GANs, they do not need to pair the generator network with any other network—neither an inference network as used with VAEs nor a discriminator network as used with GANs.

These networks are trained with a technique called *moment matching*. The basic idea behind moment matching is to train the generator in such a way that many of the statistics of samples generated by the model are as similar as possible to those of the statistics of the examples in the training set. In this context, a *moment* is an expectation of different powers of a random variable. For example, the first moment is the mean, the second moment is the mean of the squared values, and so on. In multiple dimensions, each element of the random vector may be raised to different powers, so that a moment may be any quantity of the form

$$\mathbb{E}_{\boldsymbol{x}} \Pi_i x_i^{n_i} \tag{20.82}$$

where $\boldsymbol{n} = [n_1, n_2, \ldots, n_d]^\top$ is a vector of non-negative integers.

Upon first examination, this approach seems to be computationally infeasible. For example, if we want to match all the moments of the form $x_i x_j$, then we need to minimize the difference between a number of values that is quadratic in the dimension of $\boldsymbol{x}$. Moreover, even matching all of the first and second moments would only be sufficient to fit a multivariate Gaussian distribution, which captures only linear relationships between values. Our ambitions for neural networks are to capture complex nonlinear relationships, which would require far more moments. GANs avoid this problem of exhaustively enumerating all moments by using a dynamically updated discriminator, that automatically focuses its attention on whichever statistic the generator network is matching the least effectively.

Instead, generative moment matching networks can be trained by minimizing a cost function called *maximum mean discrepancy* (Schölkopf and Smola, 2002; Gretton *et al.*, 2012) or MMD. This cost function measures the error in the first moments in an infinite-dimensional space, using an implicit mapping to feature

space defined by a kernel function in order to make computations on infinite-dimensional vectors tractable. The MMD cost is zero if and only if the two distributions being compared are equal.

Visually, the samples from generative moment matching networks are somewhat disappointing. Fortunately, they can be improved by combining the generator network with an autoencoder. First, an autoencoder is trained to reconstruct the training set. Next, the encoder of the autoencoder is used to transform the entire training set into code space. The generator network is then trained to generate code samples, which may be mapped to visually pleasing samples via the decoder.

Unlike GANs, the cost function is defined only with respect to a batch of examples from both the training set and the generator network. It is not possible to make a training update as a function of only one training example or only one sample from the generator network. This is because the moments must be computed as an empirical average across many samples. When the batch size is too small, MMD can underestimate the true amount of variation in the distributions being sampled. No finite batch size is sufficiently large to eliminate this problem entirely, but larger batches reduce the amount of underestimation. When the batch size is too large, the training procedure becomes infeasibly slow, because many examples must be processed in order to compute a single small gradient step.

As with GANs, it is possible to train a generator net using MMD even if that generator net assigns zero probability to the training points.

20.10.6 Convolutional Generative Networks

When generating images, it is often useful to use a generator network that includes a convolutional structure (see for example Goodfellow *et al.* (2014c) or Dosovitskiy *et al.* (2015)). To do so, we use the "transpose" of the convolution operator, described in Sec. 9.5. This approach often yields more realistic images and does so using fewer parameters than using fully connected layers without parameter sharing.

Convolutional networks for recognition tasks have information flow from the image to some summarization layer at the top of the network, often a class label. As this image flows upward through the network, information is discarded as the representation of the image becomes more invariant to nuisance transformations. In a generator network, the opposite is true. Rich details must be added as the representation of the image to be generated propagates through the network, culminating in the final representation of the image, which is of course the image itself, in all of its detailed glory, with object positions and poses and textures and

lighting. The primary mechanism for discarding information in a convolutional recognition network is the pooling layer. The generator network seems to need to add information. We cannot put the inverse of a pooling layer into the generator network because most pooling functions are not invertible. A simpler operation is to merely increase the spatial size of the representation. An approach that seems to perform acceptably is to use an "un-pooling" as introduced by Dosovitskiy *et al.* (2015). This layer corresponds to the inverse of the max-pooling operation under certain simplifying conditions. First, the stride of the max-pooling operation is constrained to be equal to the width of the pooling region. Second, the maximum input within each pooling region is assumed to be the input in the upper-left corner. Finally, all non-maximal inputs within each pooling region are assumed to be zero. These are very strong and unrealistic assumptions, but they do allow the max-pooling operator to be inverted. The inverse un-pooling operation allocates a tensor of zeros, then copies each value from spatial coordinate i of the input to spatial coordinate $i \times k$ of the output. The integer value k defines the size of the pooling region. Even though the assumptions motivating the definition of the un-pooling operator are unrealistic, the subsequent layers are able to learn to compensate for its unusual output, so the samples generated by the model as a whole are visually pleasing.

20.10.7 Auto-Regressive Networks

Auto-regressive networks are directed probabilistic models with no latent random variables. The conditional probability distributions in these models are represented by neural networks (sometimes extremely simple neural networks such as logistic regression). The graph structure of these models is the complete graph. They decompose a joint probability over the observed variables using the chain rule of probability to obtain a product of conditionals of the form $P(x_d \mid x_{d-1}, \ldots, x_1)$. Such models have been called *fully-visible Bayes networks* (FVBNs) and used successfully in many forms, first with logistic regression for each conditional distribution (Frey, 1998) and then with neural networks with hidden units (Bengio and Bengio, 2000b; Larochelle and Murray, 2011). In some forms of auto-regressive networks, such as NADE (Larochelle and Murray, 2011), described in Sec. 20.10.10 below, we can introduce a form of parameter sharing that brings both a statistical advantage (fewer unique parameters) and a computational advantage (less computation). This is one more instance of the recurring deep learning motif of *reuse of features*.

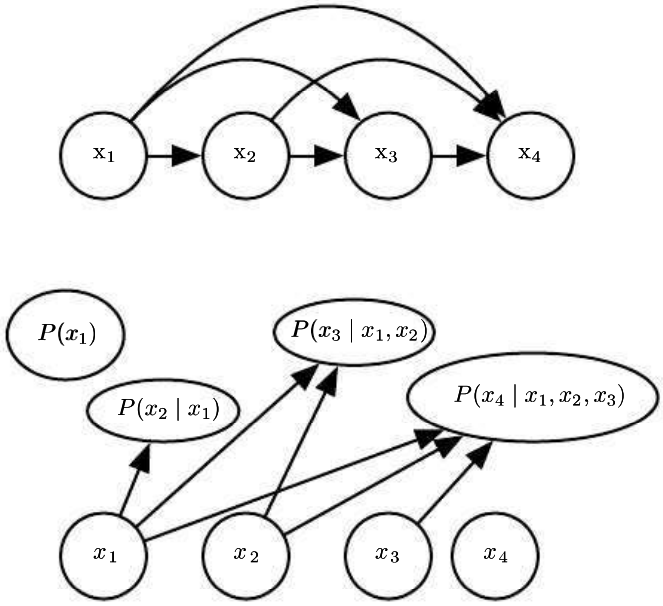

Figure 20.8: A fully visible belief network predicts the i-th variable from the $i-1$ previous ones. *(Top)* The directed graphical model for an FVBN. *(Bottom)* Corresponding computational graph, in the case of the logistic FVBN, where each prediction is made by a linear predictor.

20.10.8 Linear Auto-Regressive Networks

The simplest form of auto-regressive network has no hidden units and no sharing of parameters or features. Each $P(x_i \mid x_{i-1}, \ldots, x_1)$ is parametrized as a linear model (linear regression for real-valued data, logistic regression for binary data, softmax regression for discrete data). This model was introduced by Frey (1998) and has $O(d^2)$ parameters when there are d variables to model. It is illustrated in Fig. 20.8.

If the variables are continuous, a linear auto-regressive model is merely another way to formulate a multivariate Gaussian distribution, capturing linear pairwise interactions between the observed variables.

Linear auto-regressive networks are essentially the generalization of linear classification methods to generative modeling. They therefore have the same advantages and disadvantages as linear classifiers. Like linear classifiers, they may be trained with convex loss functions, and sometimes admit closed form solutions (as in the Gaussian case). Like linear classifiers, the model itself does not offer a way of increasing its capacity, so capacity must be raised using techniques like basis expansions of the input or the kernel trick.

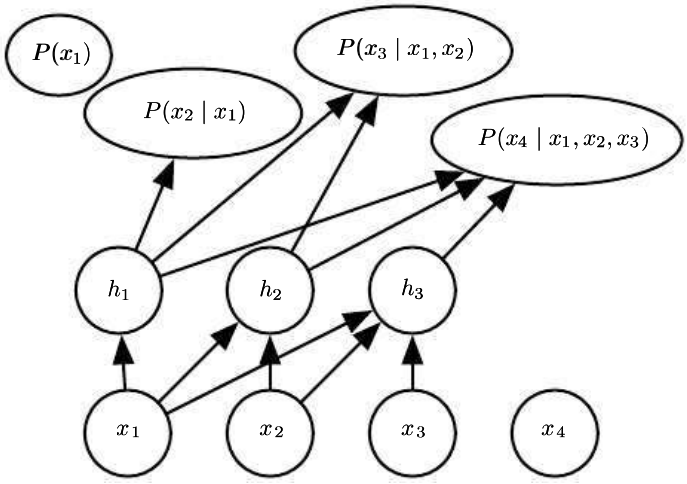

Figure 20.9: A neural auto-regressive network predicts the i-th variable x_i from the $i-1$ previous ones, but is parametrized so that features (groups of hidden units denoted h_i) that are functions of $x_1, \ldots, x_i$ can be reused in predicting all of the subsequent variables $x_{i+1}, x_{i+2}, \ldots, x_d$.

20.10.9 Neural Auto-Regressive Networks

Neural auto-regressive networks (Bengio and Bengio, 2000a,b) have the same left-to-right graphical model as logistic auto-regressive networks (Fig. 20.8) but employ a different parametrization of the conditional distributions within that graphical model structure. The new parametrization is more powerful in the sense that its capacity can be increased as much as needed, allowing approximation of any joint distribution. The new parametrization can also improve generalization by introducing a parameter sharing and feature sharing principle common to deep learning in general. The models were motivated by the objective of avoiding the curse of dimensionality arising out of traditional tabular graphical models, sharing the same structure as Fig. 20.8. In tabular discrete probabilistic models, each conditional distribution is represented by a table of probabilities, with one entry and one parameter for each possible configuration of the variables involved. By using a neural network instead, two advantages are obtained:

1. The parametrization of each $P(x_i \mid x_{i-1}, \ldots, x_1)$ by a neural network with $(i-1) \times k$ inputs and k outputs (if the variables are discrete and take k values, encoded one-hot) allows one to estimate the conditional probability without requiring an exponential number of parameters (and examples), yet still is able to capture high-order dependencies between the random variables.

2. Instead of having a different neural network for the prediction of each x_i,

a *left-to-right* connectivity illustrated in Fig. 20.9 allows one to merge all the neural networks into one. Equivalently, it means that the hidden layer features computed for predicting x_i can be reused for predicting x_{i+k} ($k > 0$). The hidden units are thus organized in *groups* that have the particularity that all the units in the i-th group only depend on the input values $x_1, \ldots, x_i$. The parameters used to compute these hidden units are jointly optimized to improve the prediction of all the variables in the sequence. This is an instance of the *reuse principle* that recurs throughout deep learning in scenarios ranging from recurrent and convolutional network architectures to multi-task and transfer learning.

Each $P(x_i \mid x_{i-1}, \ldots, x_1)$ can represent a conditional distribution by having outputs of the neural network predict *parameters* of the conditional distribution of x_i, as discussed in Sec. 6.2.1.1. Although the original neural auto-regressive networks were initially evaluated in the context of purely discrete multivariate data (with a sigmoid output for a Bernoulli variable or softmax output for a multinoulli variable) it is natural to extend such models to continuous variables or joint distributions involving both discrete and continuous variables.

20.10.10 NADE

The *neural autoregressive density estimator* (NADE) is a very successful recent form of neural auto-regressive network (Larochelle and Murray, 2011). The connectivity is the same as for the original neural auto-regressive network of Bengio and Bengio (2000b) but NADE introduces an additional parameter sharing scheme, as illustrated in Fig. 20.10. The parameters of the hidden units of different groups j are shared.

The weights $W'_{j,k,i}$ from the i-th input x_i to the k-th element of the j-th group of hidden unit $h_k^{(j)}$ ($j \geq i$) are shared among the groups:

$$W'_{j,k,i} = W_{k,i}. \tag{20.83}$$

The remaining weights, where $j < i$, are zero.

Larochelle and Murray (2011) chose this sharing scheme so that forward propagation in a NADE model loosely resembles the computations performed in mean field inference to fill in missing inputs in an RBM. This mean field inference corresponds to running a recurrent network with shared weights and the first step of that inference is the same as in NADE. The only difference is that with NADE, the output weights connecting the hidden units to the output are parametrized

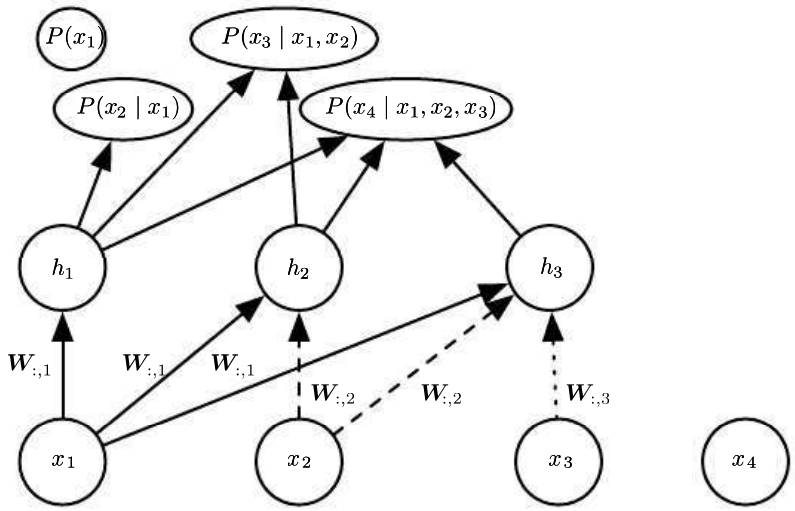

Figure 20.10: An illustration of the neural autoregressive density estimator (NADE). The hidden units are organized in groups $\boldsymbol{h}^{(j)}$ so that only the inputs $x_1, \ldots, x_i$ participate in computing $\boldsymbol{h}^{(i)}$ and predicting $P(x_j \mid x_{j-1}, \ldots, x_1)$, for $j > i$. NADE is differentiated from earlier neural auto-regressive networks by the use of a particular weight sharing pattern: $W'_{j,k,i} = W_{k,i}$ is shared (indicated in the figure by the use of the same line pattern for every instance of a replicated weight) for all the weights going out from x_i to the k-th unit of any group $j \geq i$. Recall that the vector $(W_{1,i}, W_{2,i}, \ldots, W_{n,i})$ is denoted $\boldsymbol{W}_{:,i}$.

independently from the weights connecting the input units to the hidden units. In the RBM, the hidden-to-output weights are the transpose of the input-to-hidden weights. The NADE architecture can be extended to mimic not just one time step of the mean field recurrent inference but to mimic k steps. This approach is called NADE-k (Raiko *et al.*, 2014).

As mentioned previously, auto-regressive networks may be extend to process continuous-valued data. A particularly powerful and generic way of parametrizing a continuous density is as a Gaussian mixture (introduced in Sec. 3.9.6) with mixture weights α_i (the coefficient or prior probability for component i), per-component conditional mean μ_i and per-component conditional variance σ_i^2. A model called RNADE (Uria *et al.*, 2013) uses this parametrization to extend NADE to real values. As with other mixture density networks, the parameters of this distribution are outputs of the network, with the mixture weight probabilities produced by a softmax unit, and the variances parametrized so that they are positive. Stochastic gradient descent can be numerically ill-behaved due to the interactions between the conditional means μ_i and the conditional variances σ_i^2. To reduce this difficulty, Uria *et al.* (2013) use a pseudo-gradient that replaces the gradient on the mean, in the back-propagation phase.

Another very interesting extension of the neural auto-regressive architectures gets rid of the need to choose an arbitrary *order* for the observed variables (Murray and Larochelle, 2014). In auto-regressive networks, the idea is to train the network to be able to cope with any order by randomly sampling orders and providing the information to hidden units specifying which of the inputs are observed (on the right side of the conditioning bar) and which are to be predicted and are thus considered missing (on the left side of the conditioning bar). This is nice because it allows one to use a trained auto-regressive network to *perform any inference problem* (i.e. predict or sample from the probability distribution over any subset of variables given any subset) extremely efficiently. Finally, since many orders of variables are possible ($n!$ for n variables) and each order o of variables yields a different $p(\mathbf{x} \mid o)$, we can form an ensemble of models for many values of o:

$$p_{\text{ensemble}}(\mathbf{x}) = \frac{1}{k} \sum_{i=1}^{k} p(\mathbf{x} \mid o^{(i)}). \tag{20.84}$$

This ensemble model usually generalizes better and assigns higher probability to the test set than does an individual model defined by a single ordering.

In the same paper, the authors propose deep versions of the architecture, but unfortunately that immediately makes computation as expensive as in the original neural auto-regressive neural network (Bengio and Bengio, 2000b). The first layer and the output layer can still be computed in $O(nh)$ multiply-add operations, as in the regular NADE, where h is the number of hidden units (the size of the groups h_i, in Fig. 20.10 and Fig. 20.9), whereas it is $O(n^2 h)$ in Bengio and Bengio (2000b). However, for the other hidden layers, the computation is $O(n^2 h^2)$ if every "previous" group at layer l participates in predicting the "next" group at layer $l+1$, assuming n groups of h hidden units at each layer. Making the i-th group at layer $l+1$ only depend on the i-th group, as in Murray and Larochelle (2014) at layer l reduces it to $O(nh^2)$, which is still h times worse than the regular NADE.

20.11 Drawing Samples from Autoencoders

In Chapter 14, we saw that many kinds of autoencoders learn the data distribution. There are close connections between score matching, denoising autoencoders, and contractive autoencoders. These connections demonstrate that some kinds of autoencoders learn the data distribution in some way. We have not yet seen how to draw samples from such models.

Some kinds of autoencoders, such as the variational autoencoder, explicitly

represent a probability distribution and admit straightforward ancestral sampling. Most other kinds of autoencoders require MCMC sampling.

Contractive autoencoders are designed to recover an estimate of the tangent plane of the data manifold. This means that repeated encoding and decoding with injected noise will induce a random walk along the surface of the manifold (Rifai *et al.*, 2012; Mesnil *et al.*, 2012). This manifold diffusion technique is a kind of Markov chain.

There is also a more general Markov chain that can sample from any denoising autoencoder.

20.11.1 Markov Chain Associated with any Denoising Autoencoder

The above discussion left open the question of what noise to inject and where, in order to obtain a Markov chain that would generate from the distribution estimated by the autoencoder. Bengio *et al.* (2013c) showed how to construct such a Markov chain for *generalized denoising autoencoders*. Generalized denoising autoencoders are specified by a denoising distribution for sampling an estimate of the clean input given the corrupted input.

Each step of the Markov chain that generates from the estimated distribution consists of the following sub-steps, illustrated in Fig. 20.11:

1. Starting from the previous state $\boldsymbol{x}$, inject corruption noise, sampling $\tilde{\boldsymbol{x}}$ from $C(\tilde{\boldsymbol{x}} \mid \boldsymbol{x})$.
2. Encode $\tilde{\boldsymbol{x}}$ into $\boldsymbol{h} = f(\tilde{\boldsymbol{x}})$.
3. Decode $\boldsymbol{h}$ to obtain the parameters $\boldsymbol{\omega} = g(\boldsymbol{h})$ of $p(\mathbf{x} \mid \boldsymbol{\omega} = g(\boldsymbol{h})) = p(\mathbf{x} \mid \tilde{\boldsymbol{x}})$.
4. Sample the next state $\boldsymbol{x}$ from $p(\mathbf{x} \mid \boldsymbol{\omega} = g(\boldsymbol{h})) = p(\mathbf{x} \mid \tilde{\boldsymbol{x}})$.

Bengio *et al.* (2014) showed that if the autoencoder $p(\mathbf{x} \mid \tilde{\mathbf{x}})$ forms a consistent estimator of the corresponding true conditional distribution, then the stationary distribution of the above Markov chain forms a consistent estimator (albeit an implicit one) of the data generating distribution of $\mathbf{x}$.

20.11.2 Clamping and Conditional Sampling

Similarly to Boltzmann machines, denoising autoencoders and their generalizations (such as GSNs, described below) can be used to sample from a conditional distribution $p(\mathbf{x}_f \mid \mathbf{x}_o)$, simply by clamping the *observed* units $\mathbf{x}_f$ and only resampling

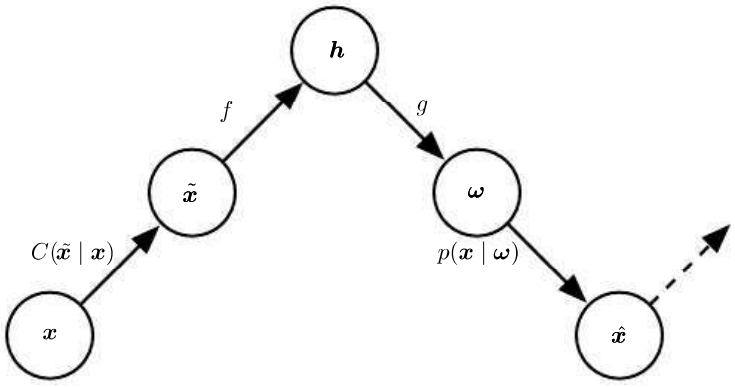

Figure 20.11: Each step of the Markov chain associated with a trained denoising autoencoder, that generates the samples from the probabilistic model implicitly trained by the denoising log-likelihood criterion. Each step consists in (a) injecting noise via corruption process C in state $\boldsymbol{x}$, yielding $\tilde{\boldsymbol{x}}$, (b) encoding it with function f, yielding $\boldsymbol{h} = f(\tilde{\boldsymbol{x}})$, (c) decoding the result with function g, yielding parameters $\boldsymbol{\omega}$ for the reconstruction distribution, and (d) given $\boldsymbol{\omega}$, sampling a new state from the reconstruction distribution $p(\mathbf{x} \mid \boldsymbol{\omega} = g(f(\tilde{\boldsymbol{x}})))$. In the typical squared reconstruction error case, $g(\boldsymbol{h}) = \hat{\boldsymbol{x}}$, which estimates $\mathbb{E}[\boldsymbol{x} \mid \tilde{\boldsymbol{x}}]$, corruption consists in adding Gaussian noise and sampling from $p(\mathbf{x} \mid \boldsymbol{\omega})$ consists in adding Gaussian noise, a second time, to the reconstruction $\hat{\boldsymbol{x}}$. The latter noise level should correspond to the mean squared error of reconstructions, whereas the injected noise is a hyperparameter that controls the mixing speed as well as the extent to which the estimator smooths the empirical distribution (Vincent, 2011). In the example illustrated here, only the C and p conditionals are stochastic steps (f and g are deterministic computations), although noise can also be injected inside the autoencoder, as in generative stochastic networks (Bengio *et al.*, 2014).

the *free* units $\mathbf{x}_o$ given $\mathbf{x}_f$ and the sampled latent variables (if any). For example, MP-DBMs can be interpreted as a form of denoising autoencoder, and are able to sample missing inputs. GSNs later generalized some of the ideas present in MP-DBMs to perform the same operation (Bengio *et al.*, 2014). Alain *et al.* (2015) identified a missing condition from Proposition 1 of Bengio *et al.* (2014), which is that the transition operator (defined by the stochastic mapping going from one state of the chain to the next) should satisfy a property called *detailed balance*, which specifies that a Markov Chain at equilibrium will remain in equilibrium whether the transition operator is run in forward or reverse.

An experiment in clamping half of the pixels (the right part of the image) and running the Markov chain on the other half is shown in Fig. 20.12.

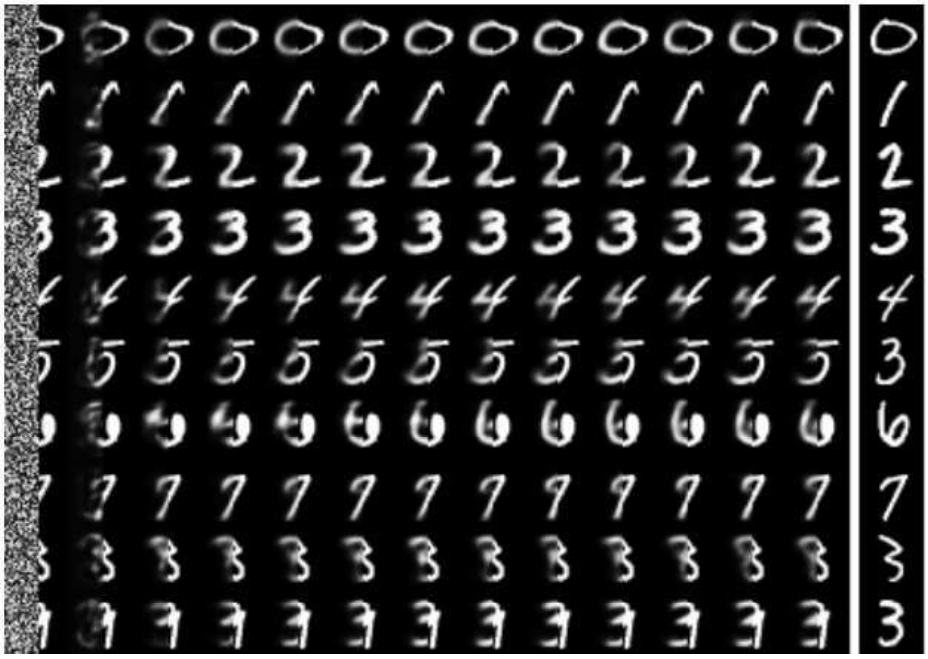

Figure 20.12: Illustration of clamping the right half of the image and running the Markov Chain by resampling only the left half at each step. These samples come from a GSN trained to reconstruct MNIST digits at each time step using the walkback procedure.

20.11.3 Walk-Back Training Procedure

The walk-back training procedure was proposed by Bengio *et al.* (2013c) as a way to accelerate the convergence of generative training of denoising autoencoders. Instead of performing a one-step encode-decode reconstruction, this procedure consists in alternative multiple stochastic encode-decode steps (as in the generative

Markov chain) initialized at a training example (just like with the contrastive divergence algorithm, described in Sec. 18.2) and penalizing the last probabilistic reconstructions (or all of the reconstructions along the way).

Training with k steps is equivalent (in the sense of achieving the same stationary distribution) as training with one step, but practically has the advantage that spurious modes farther from the data can be removed more efficiently.

20.12 Generative Stochastic Networks

Generative stochastic networks or *GSNs* (Bengio *et al.*, 2014) are generalizations of denoising autoencoders that include latent variables $\mathbf{h}$ in the generative Markov chain, in addition to the visible variables (usually denoted $\mathbf{x}$).

A GSN is parametrized by two conditional probability distributions which specify one step of the Markov chain:

1. $p(\mathbf{x}^{(k)} \mid \mathbf{h}^{(k)})$ tells how to generate the next visible variable given the current latent state. Such a "reconstruction distribution" is also found in denoising autoencoders, RBMs, DBNs and DBMs.
2. $p(\mathbf{h}^{(k)} \mid \mathbf{h}^{(k-1)}, \mathbf{x}^{(k-1)})$ tells how to update the latent state variable, given the previous latent state and visible variable.

Denoising autoencoders and GSNs differ from classical probabilistic models (directed or undirected) in that they parametrize the generative process itself rather than the mathematical specification of the joint distribution of visible and latent variables. Instead, the latter is defined **implicitly**, *if it exists*, as the stationary distribution of the generative Markov chain. The conditions for existence of the stationary distribution are mild and are the same conditions required by standard MCMC methods (see Sec. 17.3). These conditions are necessary to guarantee that the chain mixes, but they can be violated by some choices of the transition distributions (for example, if they were deterministic).

One could imagine different training criteria for GSNs. The one proposed and evaluated by Bengio *et al.* (2014) is simply reconstruction log-probability on the visible units, just like for denoising autoencoders. This is achieved by clamping $\mathbf{x}^{(0)} = \boldsymbol{x}$ to the observed example and maximizing the probability of generating $\boldsymbol{x}$ at some subsequent time steps, i.e., maximizing $\log p(\mathbf{x}^{(k)} = \boldsymbol{x} \mid \mathbf{h}^{(k)})$, where $\mathbf{h}^{(k)}$ is sampled from the chain, given $\mathbf{x}^{(0)} = \boldsymbol{x}$. In order to estimate the gradient of $\log p(\mathbf{x}^{(k)} = \boldsymbol{x} \mid \mathbf{h}^{(k)})$ with respect to the other pieces of the model, Bengio *et al.* (2014) use the reparametrization trick, introduced in Sec. 20.9.

The *walk-back training* protocol (described in Sec. 20.11.3) was used (Bengio *et al.*, 2014) to improve training convergence of GSNs.

20.12.1 Discriminant GSNs

The original formulation of GSNs (Bengio *et al.*, 2014) was meant for unsupervised learning and implicitly modeling $p(\mathbf{x})$ for observed data $\mathbf{x}$, but it is possible to modify the framework to optimize $p(\mathbf{y} \mid \boldsymbol{x})$.

For example, Zhou and Troyanskaya (2014) generalize GSNs in this way, by only back-propagating the reconstruction log-probability over the output variables, keeping the input variables fixed. They applied this successfully to model **sequences** (protein secondary structure) and introduced a (one-dimensional) **convolutional** structure in the transition operator of the Markov chain. It is important to remember that, for each step of the Markov chain, one generates a new sequence for each layer, and that sequence is the input for computing other layer values (say the one below and the one above) at the next time step.

Hence the Markov chain is really over the **output variable** (and associated higher-level hidden layers), and the input sequence only serves to condition that chain, with back-propagation allowing to learn how the input sequence can condition the output distribution implicitly represented by the Markov chain. It is therefore a case of using the GSN in the context of **structured outputs**, where $p(\mathbf{y} \mid \mathbf{x})$ does not have a simple parametric form but instead the components of $\mathbf{y}$ are statistically dependent of each other, given $\mathbf{x}$, in complicated ways.

Zöhrer and Pernkopf (2014) introduced a hybrid model that combines a supervised objective (as in the above work) and an unsupervised objective (as in the original GSN work), by simply adding (with a different weight) the supervised and unsupervised costs i.e., the reconstruction log-probabilities of $\mathbf{y}$ and $\mathbf{x}$ respectively. Such a hybrid criterion had previously been introduced for RBMs by Larochelle and Bengio (2008). They show improved classification performance using this scheme.

20.13 Other Generation Schemes

The methods we have described so far use either MCMC sampling, ancestral sampling, or some mixture of the two to generate samples. While these are the most popular approaches to generative modeling, they are by no means the only approaches.

Sohl-Dickstein *et al.* (2015) developed a *diffusion inversion* training scheme for learning a generative model, based on non-equilibrium thermodynamics. The approach is based on the idea that the probability distributions we wish to sample from have structure. This structure can gradually be destroyed by a diffusion process that incrementally changes the probability distribution to have more entropy. To form a generative model, we can run the process in reverse, by training a model that gradually restores the structure to an unstructured distribution. By iteratively applying a process that brings a distribution closer to the target one, we can gradually approach that target distribution. This approach resembles MCMC methods in the sense that it involves many iterations to produce a sample. However, the model is defined to be the probability distribution produced by the final step of the chain. In this sense, there is no approximation induced by the iterative procedure. The approach introduced by Sohl-Dickstein *et al.* (2015) is also very close to the generative interpretation of the denoising autoencoder (Sec. 20.11.1). Like with the denoising autoencoder, the training objective trains a transition operator which attempts to probabilistically undo the effect of adding some noise, trying to undo one step of the diffusion process. If we compare with the walkback training procedure (Sec. 20.11.3) for denoising autoencoders and GSNs, the main difference is that instead of reconstructing only towards the observed training point $\boldsymbol{x}$, the objective function only tries to reconstruct towards the previous point in the diffusion trajectory that started at $\boldsymbol{x}$ (which should be easier). This addresses the following dilemma present with the ordinary reconstruction log-likelihood objective of denoising autoencoders: with small levels of noise the learner only sees configurations near the data points, while with large levels of noise it is asked to do an almost impossible job (because the denoising distribution is going to be highly complex and multi-modal). With the diffusion inversion objective, the learner can learn more precisely the shape of the density around the data points as well as remove spurious modes that could show up far from the data points.

Another approach to sample generation is the *approximate Bayesian computation* (ABC) framework (Rubin *et al.*, 1984). In this approach, samples are rejected or modified in order to make the moments of selected functions of the samples match those of the desired distribution. While this idea uses the moments of the samples like in moment matching, it is different from moment matching because it modifies the samples themselves, rather than training the model to automatically emit samples with the correct moments. Bachman and Precup (2015) showed how to use ideas from ABC in the context of deep learning, by using ABC to shape the MCMC trajectories of GSNs.

We expect that many other possible approaches to generative modeling await discovery.

20.14 Evaluating Generative Models

Researchers studying generative models often need to compare one generative model to another, usually in order to demonstrate that a newly invented generative model is better at capturing some distribution than the pre-existing models.

This can be a difficult and subtle task. In many cases, we can not actually evaluate the log probability of the data under the model, but only an approximation. In these cases, it is important to think and communicate clearly about exactly what is being measured. For example, suppose we can evaluate a stochastic estimate of the log-likelihood for model A, and a deterministic lower bound on the log-likelihood for model B. If model A gets a higher score than model B, which is better? If we care about determining which model has a better internal representation of the distribution, we actually cannot tell, unless we have some way of determining how loose the bound for model B is. However, if we care about how well we can use the model in practice, for example to perform anomaly detection, then it is fair to say that a model is preferable based on a criterion specific to the practical task of interest, e.g., based on ranking test examples and ranking criteria such as precision and recall.

Another subtlety of evaluating generative models is that the evaluation metrics are often hard research problems in and of themselves. It can be very difficult to establish that models are being compared fairly. For example, suppose we use AIS to estimate $\log Z$ in order to compute $\log \tilde{p}(\boldsymbol{x}) - \log Z$ for a new model we have just invented. A computationally economical implementation of AIS may fail to find several modes of the model distribution and underestimate Z, which will result in us overestimating $\log p(\boldsymbol{x})$. It can thus be difficult to tell whether a high likelihood estimate is due to a good model or a bad AIS implementation.

Other fields of machine learning usually allow for some variation in the preprocessing of the data. For example, when comparing the accuracy of object recognition algorithms, it is usually acceptable to preprocess the input images slightly differently for each algorithm based on what kind of input requirements it has. Generative modeling is different because changes in preprocessing, even very small and subtle ones, are completely unacceptable. Any change to the input data changes the distribution to be captured and fundamentally alters the task. For example, multiplying the input by 0.1 will artificially increase likelihood by a factor of 10.

Issues with preprocessing commonly arise when benchmarking generative models on the MNIST dataset, one of the more popular generative modeling benchmarks. MNIST consists of grayscale images. Some models treat MNIST images as points

in a real vector space, while others treat them as binary. Yet others treat the grayscale values as probabilities for a binary samples. It is essential to compare real-valued models only to other real-valued models and binary-valued models only to other binary-valued models. Otherwise the likelihoods measured are not on the same space. For binary-valued models, the log-likelihood can be at most zero, while for real-valued models it can be arbitrarily high, since it is the measurement of a density. Among binary models, it is important to compare models using exactly the same kind of binarization. For example, we might binarize a gray pixel to 0 or 1 by thresholding at 0.5, or by drawing a random sample whose probability of being 1 is given by the gray pixel intensity. If we use the random binarization, we might binarize the whole dataset once, or we might draw a different random example for each step of training and then draw multiple samples for evaluation. Each of these three schemes yields wildly different likelihood numbers, and when comparing different models it is important that both models use the same binarization scheme for training and for evaluation. In fact, researchers who apply a single random binarization step share a file containing the results of the random binarization, so that there is no difference in results based on different outcomes of the binarization step.

Because being able to generate realistic samples from the data distribution is one of the goals of a generative model, practitioners often evaluate generative models by visually inspecting the samples. In the best case, this is done not by the researchers themselves, but by experimental subjects who do not know the source of the samples (Denton *et al.*, 2015). Unfortunately, it is possible for a very poor probabilistic model to produce very good samples. A common practice to verify if the model only copies some of the training examples is illustrated in Fig. 16.1. The idea is to show for some of the generated samples their nearest neighbor in the training set, according to Euclidean distance in the space of $\boldsymbol{x}$. This test is intended to detect the case where the model overfits the training set and just reproduces training instances. It is even possible to simultaneously underfit and overfit yet still produce samples that individually look good. Imagine a generative model trained on images of dogs and cats that simply learns to reproduce the training images of dogs. Such a model has clearly overfit, because it does not produces images that were not in the training set, but it has also underfit, because it assigns no probability to the training images of cats. Yet a human observer would judge each individual image of a dog to be high quality. In this simple example, it would be easy for a human observer who can inspect many samples to determine that the cats are absent. In more realistic settings, a generative model trained on data with tens of thousands of modes may ignore a small number of modes, and a human observer would not easily be able to inspect or remember

enough images to detect the missing variation.

Since the visual quality of samples is not a reliable guide, we often also evaluate the log-likelihood that the model assigns to the test data, when this is computationally feasible. Unfortunately, in some cases the likelihood seems not to measure any attribute of the model that we really care about. For example, real-valued models of MNIST can obtain arbitrarily high likelihood by assigning arbitrarily low variance to background pixels that never change. Models and algorithms that detect these constant features can reap unlimited rewards, even though this is not a very useful thing to do. The potential to achieve a cost approaching negative infinity is present for any kind of maximum likelihood problem with real values, but it is especially problematic for generative models of MNIST because so many of the output values are trivial to predict. This strongly suggests a need for developing other ways of evaluating generative models.

Theis *et al.* (2015) review many of the issues involved in evaluating generative models, including many of the ideas described above. They highlight the fact that there are many different uses of generative models and that the choice of metric must match the intended use of the model. For example, some generative models are better at assigning high probability to most realistic points while other generative models are better at rarely assigning high probability to unrealistic points. These differences can result from whether a generative model is designed to minimize $D_{\mathrm{KL}}(p_{\mathrm{data}} \| p_{\mathrm{model}})$ or $D_{\mathrm{KL}}(p_{\mathrm{model}} \| p_{\mathrm{data}})$, as illustrated in Fig. 3.6. Unfortunately, even when we restrict the use of each metric to the task it is most suited for, all of the metrics currently in use continue to have serious weaknesses. One of the most important research topics in generative modeling is therefore not just how to improve generative models, but in fact, designing new techniques to measure our progress.

20.15 Conclusion

Training generative models with hidden units is a powerful way to make models understand the world represented in the given training data. By learning a model $p_{\mathrm{model}}(\boldsymbol{x})$ and a representation $p_{\mathrm{model}}(\boldsymbol{h} \mid \boldsymbol{x})$, a generative model can provide answers to many inference problems about the relationships between input variables in $\boldsymbol{x}$ and can provide many different ways of representing $\boldsymbol{x}$ by taking expectations of $\boldsymbol{h}$ at different layers of the hierarchy. Generative models hold the promise to provide AI systems with a framework for all of the many different intuitive concepts they need to understand, and the ability to reason about these concepts in the face of uncertainty. We hope that our readers will find new ways to make these

approaches more powerful and continue the journey to understanding the principles that underlie learning and intelligence.