Chapter 20

Deep Generative Models

In this chapter, we present several of the specific kinds of generative models that can be built and trained using the techniques presented in chapters16–19. All these models represent probability distributions over multiple variables in some way. Some allow the probability distribution function to be evaluated explicitly. Others do not allow the evaluation of the probability distribution function but support operations that implicitly require knowledge of it, such as drawing samples from the distribution. Some of these models are structured probabilistic models described in terms of graphs and factors, using the language of graphical models presented in chapter 16. Others cannot be easily described in terms of factors but represent probability distributions nonetheless.

20.1 Boltzmann Machines

Boltzmann machines were originally introduced as a general "connectionist" approach to learning arbitrary probability distributions over binary vectors (Fahlman *et al.*, 1983; Ackley *et al.*, 1985; Hinton *et al.*, 1984; Hinton and Sejnowski, 1986). Variants of the Boltzmann machine that include other kinds of variables have long ago surpassed the popularity of the original. In this section we briefly introduce the binary Boltzmann machine and discuss the issues that come up when trying to train and perform inference in the model.

We define the Boltzmann machine over a d-dimensional binary random vector $\mathbf{x} \in \{0, 1\}^d$. The Boltzmann machine is an energy-based model (section 16.2.4),

meaning we define the joint probability distribution using an energy function:

$$P(\boldsymbol{x}) = \frac{\exp\left(-E(\boldsymbol{x})\right)}{Z}, \tag{20.1}$$

where $E(\boldsymbol{x})$ is the energy function, and Z is the partition function that ensures that $\sum_{\boldsymbol{x}} P(\boldsymbol{x}) = 1$. The energy function of the Boltzmann machine is given by

$$E(\boldsymbol{x}) = -\boldsymbol{x}^\top \boldsymbol{U}\boldsymbol{x} - \boldsymbol{b}^\top \boldsymbol{x}, \tag{20.2}$$

where $\boldsymbol{U}$ is the "weight" matrix of model parameters and $\boldsymbol{b}$ is the vector of bias parameters.

In the general setting of the Boltzmann machine, we are given a set of training examples, each of which are n-dimensional. Equation 20.1 describes the joint probability distribution over the observed variables. While this scenario is certainly viable, it does limit the kinds of interactions between the observed variables to those described by the weight matrix. Specifically, it means that the probability of one unit being on is given by a linear model (logistic regression) from the values of the other units.

The Boltzmann machine becomes more powerful when not all the variables are observed. In this case, the latent variables can act similarly to hidden units in a multilayer perceptron and model higher-order interactions among the visible units. Just as the addition of hidden units to convert logistic regression into an MLP results in the MLP being a universal approximator of functions, a Boltzmann machine with hidden units is no longer limited to modeling linear relationships between variables. Instead, the Boltzmann machine becomes a universal approximator of probability mass functions over discrete variables (Le Roux and Bengio, 2008).

Formally, we decompose the units $\boldsymbol{x}$ into two subsets: the visible units $\boldsymbol{v}$ and the latent (or hidden) units $\boldsymbol{h}$. The energy function becomes

$$E(\boldsymbol{v}, \boldsymbol{h}) = -\boldsymbol{v}^\top \boldsymbol{R}\boldsymbol{v} - \boldsymbol{v}^\top \boldsymbol{W}\boldsymbol{h} - \boldsymbol{h}^\top \boldsymbol{S}\boldsymbol{h} - \boldsymbol{b}^\top \boldsymbol{v} - \boldsymbol{c}^\top \boldsymbol{h}. \tag{20.3}$$

Boltzmann Machine Learning Learning algorithms for Boltzmann machines are usually based on maximum likelihood. All Boltzmann machines have an intractable partition function, so the maximum likelihood gradient must be approximated using the techniques described in chapter 18.

One interesting property of Boltzmann machines when trained with learning rules based on maximum likelihood is that the update for a particular weight connecting two units depends only on the statistics of those two units, collected under different distributions: $P_{\text{model}}(\boldsymbol{v})$ and $\hat{P}_{\text{data}}(\boldsymbol{v}) P_{\text{model}}(\boldsymbol{h} \mid \boldsymbol{v})$. The rest of the

network participates in shaping those statistics, but the weight can be updated without knowing anything about the rest of the network or how those statistics were produced. This means that the learning rule is "local," which makes Boltzmann machine learning somewhat biologically plausible. It is conceivable that if each neuron were a random variable in a Boltzmann machine, then the axons and dendrites connecting two random variables could learn only by observing the firing pattern of the cells that they actually physically touch. In particular, in the positive phase, two units that frequently activate together have their connection strengthened. This is an example of a Hebbian learning rule (Hebb, 1949) often summarized with the mnemonic "fire together, wire together." Hebbian learning rules are among the oldest hypothesized explanations for learning in biological systems and remain relevant today (Giudice *et al.*, 2009).

Other learning algorithms that use more information than local statistics seem to require us to hypothesize the existence of more machinery than this. For example, for the brain to implement back-propagation in a multilayer perceptron, it seems necessary for the brain to maintain a secondary communication network for transmitting gradient information backward through the network. Proposals for biologically plausible implementations (and approximations) of back-propagation have been made (Hinton, 2007a; Bengio, 2015) but remain to be validated, and Bengio (2015) links back-propagation of gradients to inference in energy-based models similar to the Boltzmann machine (but with continuous latent variables).

The negative phase of Boltzmann machine learning is somewhat harder to explain from a biological point of view. As argued in section 18.2, dream sleep may be a form of negative phase sampling. This idea is more speculative though.

20.2 Restricted Boltzmann Machines

Invented under the name **harmonium** (Smolensky, 1986), restricted Boltzmann machines are some of the most common building blocks of deep probabilistic models. We briefly describe RBMs in section 16.7.1. Here we review the previous information and go into more detail. RBMs are undirected probabilistic graphical models containing a layer of observable variables and a single layer of latent variables. RBMs may be stacked (one on top of the other) to form deeper models. See figure 20.1 for some examples. In particular, figure 20.1a shows the graph structure of the RBM itself. It is a bipartite graph, with no connections permitted between any variables in the observed layer or between any units in the latent layer.

We begin with the binary version of the restricted Boltzmann machine, but as

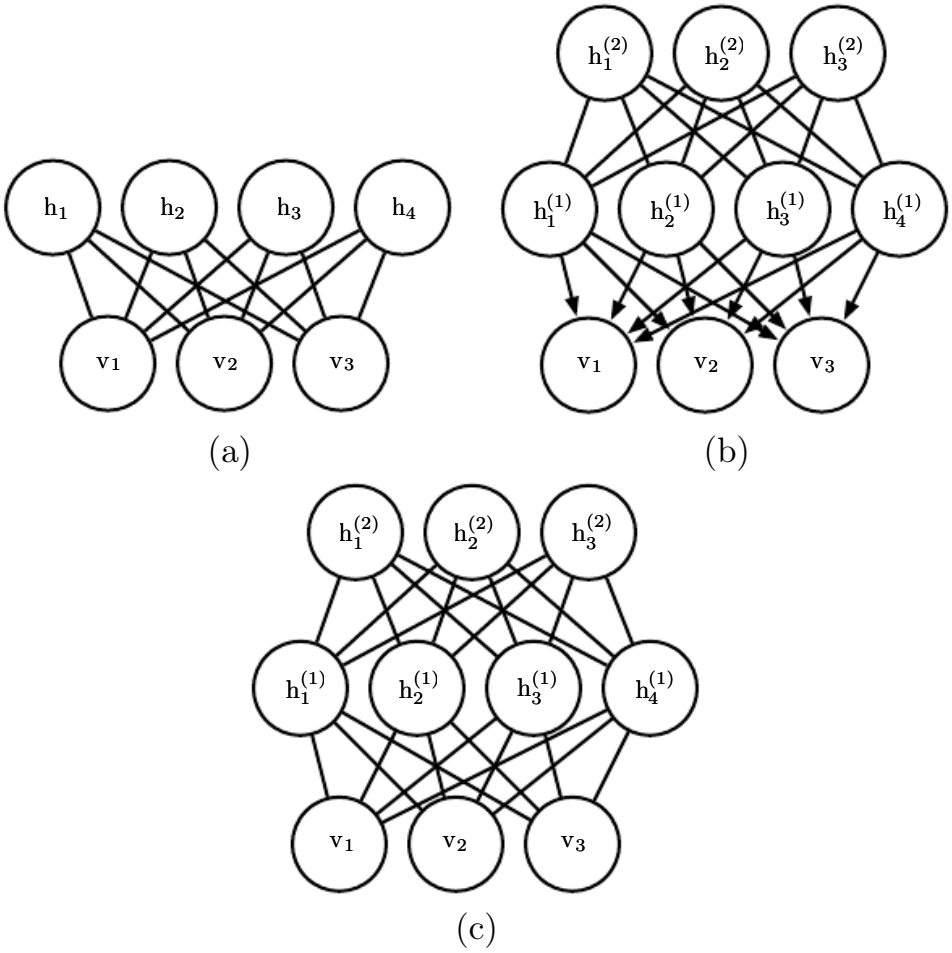

Figure 20.1: Examples of models that may be built with restricted Boltzmann machines. *(a)* The restricted Boltzmann machine itself is an undirected graphical model based on a bipartite graph, with visible units in one part of the graph and hidden units in the other part. There are no connections among the visible units, nor any connections among the hidden units. Typically every visible unit is connected to every hidden unit, but it is possible to construct sparsely connected RBMs such as convolutional RBMs. *(b)* A deep belief network is a hybrid graphical model involving both directed and undirected connections. Like an RBM, it has no intralayer connections. However, a DBN has multiple hidden layers, and thus connections between hidden units that are in separate layers. All the local conditional probability distributions needed by the deep belief network are copied directly from the local conditional probability distributions of its constituent RBMs. Alternatively, we could also represent the deep belief network with a completely undirected graph, but it would need intralayer connections to capture the dependencies between parents. *(c)* A deep Boltzmann machine is an undirected graphical model with several layers of latent variables. Like RBMs and DBNs, DBMs lack intralayer connections. DBMs are less closely tied to RBMs than DBNs are. When initializing a DBM from a stack of RBMs, it is necessary to modify the RBM parameters slightly. Some kinds of DBMs may be trained without first training a set of RBMs.

we see later, there are extensions to other types of visible and hidden units.

More formally, let the observed layer consist of a set of n_v binary random variables, which we refer to collectively with the vector $\mathbf{v}$. We refer to the latent, or hidden, layer of n_h binary random variables as $\boldsymbol{h}$.

Like the general Boltzmann machine, the restricted Boltzmann machine is an energy-based model with the joint probability distribution specified by its energy function:

$$P(\mathbf{v} = \boldsymbol{v}, \mathbf{h} = \boldsymbol{h}) = \frac{1}{Z} \exp\left(-E(\boldsymbol{v}, \boldsymbol{h})\right). \tag{20.4}$$

The energy function for an RBM is given by

$$E(\boldsymbol{v}, \boldsymbol{h}) = -\boldsymbol{b}^\top \boldsymbol{v} - \boldsymbol{c}^\top \boldsymbol{h} - \boldsymbol{v}^\top \boldsymbol{W} \boldsymbol{h}, \tag{20.5}$$

and Z is the normalizing constant known as the partition function:

$$Z = \sum_{\boldsymbol{v}} \sum_{\boldsymbol{h}} \exp\left\{-E(\boldsymbol{v}, \boldsymbol{h})\right\}. \tag{20.6}$$

It is apparent from the definition of the partition function Z that the naive method of computing Z (exhaustively summing over all states) could be computationally intractable, unless a cleverly designed algorithm could exploit regularities in the probability distribution to compute Z faster. In the case of restricted Boltzmann machines, Long and Servedio (2010) formally proved that the partition function Z is intractable. The intractable partition function Z implies that the normalized joint probability distribution $P(\boldsymbol{v})$ is also intractable to evaluate.

20.2.1 Conditional Distributions

Though $P(\boldsymbol{v})$ is intractable, the bipartite graph structure of the RBM has the special property of its conditional distributions $P(\mathbf{h} \mid \mathbf{v})$ and $P(\mathbf{v} \mid \mathbf{h})$ being factorial and relatively simple to compute and sample from.

Deriving the conditional distributions from the joint distribution is straightforward:

$$P(\boldsymbol{h} \mid \boldsymbol{v}) = \frac{P(\boldsymbol{h}, \boldsymbol{v})}{P(\boldsymbol{v})} \tag{20.7}$$

$$= \frac{1}{P(\boldsymbol{v})} \frac{1}{Z} \exp\left\{\boldsymbol{b}^\top \boldsymbol{v} + \boldsymbol{c}^\top \boldsymbol{h} + \boldsymbol{v}^\top \boldsymbol{W} \boldsymbol{h}\right\} \tag{20.8}$$

$$= \frac{1}{Z'} \exp\left\{\boldsymbol{c}^\top \boldsymbol{h} + \boldsymbol{v}^\top \boldsymbol{W} \boldsymbol{h}\right\} \tag{20.9}$$

$$= \frac{1}{Z'} \exp\left\{ \sum_{j=1}^{n_h} c_j h_j + \sum_{j=1}^{n_h} \boldsymbol{v}^\top \boldsymbol{W}_{:,j} \boldsymbol{h}_j \right\} \tag{20.10}$$

$$= \frac{1}{Z'} \prod_{j=1}^{n_h} \exp\left\{ c_j h_j + \boldsymbol{v}^\top \boldsymbol{W}_{:,j} \boldsymbol{h}_j \right\}. \tag{20.11}$$

Since we are conditioning on the visible units $\mathbf{v}$, we can treat these as constant with respect to the distribution $P(\mathbf{h} \mid \mathbf{v})$. The factorial nature of the conditional $P(\mathbf{h} \mid \mathbf{v})$ follows immediately from our ability to write the joint probability over the vector $\boldsymbol{h}$ as the product of (unnormalized) distributions over the individual elements, h_j. It is now a simple matter of normalizing the distributions over the individual binary h_j.

$$P(h_j = 1 \mid \boldsymbol{v}) = \frac{\tilde{P}(h_j = 1 \mid \boldsymbol{v})}{\tilde{P}(h_j = 0 \mid \boldsymbol{v}) + \tilde{P}(h_j = 1 \mid \boldsymbol{v})} \tag{20.12}$$

$$= \frac{\exp\left\{ c_j + \boldsymbol{v}^\top \boldsymbol{W}_{:,j} \right\}}{\exp\left\{0\right\} + \exp\left\{ c_j + \boldsymbol{v}^\top \boldsymbol{W}_{:,j} \right\}} \tag{20.13}$$

$$= \sigma\left(c_j + \boldsymbol{v}^\top \boldsymbol{W}_{:,j} \right). \tag{20.14}$$

We can now express the full conditional over the hidden layer as the factorial distribution:

$$P(\boldsymbol{h} \mid \boldsymbol{v}) = \prod_{j=1}^{n_h} \sigma\left((2\boldsymbol{h} - 1) \odot (\boldsymbol{c} + \boldsymbol{W}^\top \boldsymbol{v}) \right)_j. \tag{20.15}$$

A similar derivation will show that the other condition of interest to us, $P(\boldsymbol{v} \mid \boldsymbol{h})$, is also a factorial distribution:

$$P(\boldsymbol{v} \mid \boldsymbol{h}) = \prod_{i=1}^{n_v} \sigma\left((2\boldsymbol{v} - 1) \odot (\boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}) \right)_i. \tag{20.16}$$

20.2.2 Training Restricted Boltzmann Machines

Because the RBM admits efficient evaluation and differentiation of $\tilde{P}(\boldsymbol{v})$ and efficient MCMC sampling in the form of block Gibbs sampling, it can readily be trained with any of the techniques described in chapter 18 for training models that have intractable partition functions. This includes CD, SML (PCD), ratio matching, and so on. Compared to other undirected models used in deep learning, the RBM is relatively straightforward to train because we can compute $P(\mathbf{h} \mid \boldsymbol{v})$

exactly in closed form. Some other deep models, such as the deep Boltzmann machine, combine both the difficulty of an intractable partition function and the difficulty of intractable inference.

20.3 Deep Belief Networks

Deep belief networks (DBNs) were one of the first nonconvolutional models to successfully admit training of deep architectures (Hinton *et al.*, 2006; Hinton, 2007b). The introduction of deep belief networks in 2006 began the current deep learning renaissance. Prior to the introduction of deep belief networks, deep models were considered too difficult to optimize. Kernel machines with convex objective functions dominated the research landscape. Deep belief networks demonstrated that deep architectures can be successful by outperforming kernelized support vector machines on the MNIST dataset (Hinton *et al.*, 2006). Today, deep belief networks have mostly fallen out of favor and are rarely used, even compared to other unsupervised or generative learning algorithms, but they are still deservedly recognized for their important role in deep learning history.

Deep belief networks are generative models with several layers of latent variables. The latent variables are typically binary, while the visible units may be binary or real. There are no intralayer connections. Usually, every unit in each layer is connected to every unit in each neighboring layer, though it is possible to construct more sparsely connected DBNs. The connections between the top two layers are undirected. The connections between all other layers are directed, with the arrows pointed toward the layer that is closest to the data. See figure 20.1b for an example.

A DBN with l hidden layers contains l weight matrices: $\boldsymbol{W}^{(1)}, \ldots, \boldsymbol{W}^{(l)}$. It also contains $l+1$ bias vectors $\boldsymbol{b}^{(0)}, \ldots, \boldsymbol{b}^{(l)}$, with $\boldsymbol{b}^{(0)}$ providing the biases for the visible layer. The probability distribution represented by the DBN is given by

$$P(\boldsymbol{h}^{(l)}, \boldsymbol{h}^{(l-1)}) \propto \exp\left(\boldsymbol{b}^{(l)\top}\boldsymbol{h}^{(l)} + \boldsymbol{b}^{(l-1)\top}\boldsymbol{h}^{(l-1)} + \boldsymbol{h}^{(l-1)\top}\boldsymbol{W}^{(l)}\boldsymbol{h}^{(l)}\right), \tag{20.17}$$

$$P(h_i^{(k)} = 1 \mid \boldsymbol{h}^{(k+1)}) = \sigma\left(b_i^{(k)} + \boldsymbol{W}_{:,i}^{(k+1)\top}\boldsymbol{h}^{(k+1)}\right) \forall i, \forall k \in 1, \ldots, l-2, \tag{20.18}$$

$$P(v_i = 1 \mid \boldsymbol{h}^{(1)}) = \sigma\left(b_i^{(0)} + \boldsymbol{W}_{:,i}^{(1)\top}\boldsymbol{h}^{(1)}\right) \forall i. \tag{20.19}$$

In the case of real-valued visible units, substitute

$$\mathbf{v} \sim \mathcal{N}\left(\boldsymbol{v}; \boldsymbol{b}^{(0)} + \boldsymbol{W}^{(1)\top}\boldsymbol{h}^{(1)}, \boldsymbol{\beta}^{-1}\right) \tag{20.20}$$

with $\boldsymbol{\beta}$ diagonal for tractability. Generalizations to other exponential family visible units are straightforward, at least in theory. A DBN with only one hidden layer is just an RBM.

To generate a sample from a DBN, we first run several steps of Gibbs sampling on the top two hidden layers. This stage is essentially drawing a sample from the RBM defined by the top two hidden layers. We can then use a single pass of ancestral sampling through the rest of the model to draw a sample from the visible units.

Deep belief networks incur many of the problems associated with both directed models and undirected models.

Inference in a deep belief network is intractable because of the explaining away effect within each directed layer and the interaction between the two hidden layers that have undirected connections. Evaluating or maximizing the standard evidence lower bound on the log-likelihood is also intractable, because the evidence lower bound takes the expectation of cliques whose size is equal to the network width.

Evaluating or maximizing the log-likelihood requires confronting not just the problem of intractable inference to marginalize out the latent variables, but also the problem of an intractable partition function within the undirected model of the top two layers.

To train a deep belief network, one begins by training an RBM to maximize $\mathbb{E}_{\mathbf{v} \sim p_{\text{data}}} \log p(\boldsymbol{v})$ using contrastive divergence or stochastic maximum likelihood. The parameters of the RBM then define the parameters of the first layer of the DBN. Next, a second RBM is trained to approximately maximize

$$\mathbb{E}_{\mathbf{v} \sim p_{\text{data}}} \mathbb{E}_{\mathbf{h}^{(1)} \sim p^{(1)}(\boldsymbol{h}^{(1)} | \boldsymbol{v})} \log p^{(2)}(\boldsymbol{h}^{(1)}), \tag{20.21}$$

where $p^{(1)}$ is the probability distribution represented by the first RBM, and $p^{(2)}$ is the probability distribution represented by the second RBM. In other words, the second RBM is trained to model the distribution defined by sampling the hidden units of the first RBM, when the first RBM is driven by the data. This procedure can be repeated indefinitely, to add as many layers to the DBN as desired, with each new RBM modeling the samples of the previous one. Each RBM defines another layer of the DBN. This procedure can be justified as increasing a variational lower bound on the log-likelihood of the data under the DBN (Hinton *et al.*, 2006).

In most applications, no effort is made to jointly train the DBN after the greedy layer-wise procedure is complete. However, it is possible to perform generative fine-tuning using the wake-sleep algorithm.

The trained DBN may be used directly as a generative model, but most of the interest in DBNs arose from their ability to improve classification models. We can take the weights from the DBN and use them to define an MLP:

$$\boldsymbol{h}^{(1)} = \sigma\left(\boldsymbol{b}^{(1)} + \boldsymbol{v}^\top \boldsymbol{W}^{(1)}\right), \tag{20.22}$$

$$\boldsymbol{h}^{(l)} = \sigma\left(\boldsymbol{b}^{(l)} + \boldsymbol{h}^{(l-1)\top} \boldsymbol{W}^{(l)}\right) \forall l \in 2, \ldots, m. \tag{20.23}$$

After initializing this MLP with the weights and biases learned via generative training of the DBN, we can train the MLP to perform a classification task. This additional training of the MLP is an example of discriminative fine-tuning.

This specific choice of MLP is somewhat arbitrary, compared to many of the inference equations in chapter 19 that are derived from first principles. This MLP is a heuristic choice that seems to work well in practice and is used consistently in the literature. Many approximate inference techniques are motivated by their ability to find a maximally *tight* variational lower bound on the log-likelihood under some set of constraints. One can construct a variational lower bound on the log-likelihood using the hidden unit expectations defined by the DBN's MLP, but this is true of *any* probability distribution over the hidden units, and there is no reason to believe that this MLP provides a particularly tight bound. In particular, the MLP ignores many important interactions in the DBN graphical model. The MLP propagates information upward from the visible units to the deepest hidden units, but it does not propagate any information downward or sideways. The DBN graphical model has explaining away interactions between all the hidden units within the same layer as well as in top-down interactions between layers.

While the log-likelihood of a DBN is intractable, it may be approximated with AIS (Salakhutdinov and Murray, 2008). This permits evaluating its quality as a generative model.

The term "deep belief network" is commonly used incorrectly to refer to any kind of deep neural network, even networks without latent variable semantics. The term should refer specifically to models with undirected connections in the deepest layer and directed connections pointing downward between all other pairs of consecutive layers.

The term may also cause some confusion because "belief network" is sometimes used to refer to purely directed models, while deep belief networks contain an undirected layer. Deep belief networks also share the acronym DBN with dynamic Bayesian networks (Dean and Kanazawa, 1989), which are Bayesian networks for representing Markov chains.

20.4 Deep Boltzmann Machines

A **deep Boltzmann machine**, or DBM (Salakhutdinov and Hinton, 2009a) is another kind of deep generative model. Unlike the deep belief network (DBN), it is an entirely undirected model. Unlike the RBM, the DBM has several layers of latent variables (RBMs have just one). But like the RBM, within each layer, each of the variables are mutually independent, conditioned on the variables in the neighboring layers. See figure 20.2 for the graph structure. Deep Boltzmann machines have been applied to a variety of tasks, including document modeling (Srivastava *et al.*, 2013).

Like RBMs and DBNs, DBMs typically contain only binary units—as we assume for simplicity of our presentation of the model—but it is straightforward to include real-valued visible units.

A DBM is an energy-based model, meaning that the joint probability distribution over the model variables is parametrized by an energy function E. In the case of a deep Boltzmann machine with one visible layer, $\boldsymbol{v}$, and three hidden layers, $\boldsymbol{h}^{(1)}$, $\boldsymbol{h}^{(2)}$, and $\boldsymbol{h}^{(3)}$, the joint probability is given by

$$P\left(\boldsymbol{v}, \boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)}, \boldsymbol{h}^{(3)}\right) = \frac{1}{Z(\boldsymbol{\theta})} \exp\left(-E(\boldsymbol{v}, \boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)}, \boldsymbol{h}^{(3)}; \boldsymbol{\theta})\right). \tag{20.24}$$

To simplify our presentation, we omit the bias parameters below. The DBM energy function is then defined as follows:

$$E(\boldsymbol{v}, \boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)}, \boldsymbol{h}^{(3)}; \boldsymbol{\theta}) = -\boldsymbol{v}^\top \boldsymbol{W}^{(1)} \boldsymbol{h}^{(1)} - \boldsymbol{h}^{(1)\top} \boldsymbol{W}^{(2)} \boldsymbol{h}^{(2)} - \boldsymbol{h}^{(2)\top} \boldsymbol{W}^{(3)} \boldsymbol{h}^{(3)}. \tag{20.25}$$

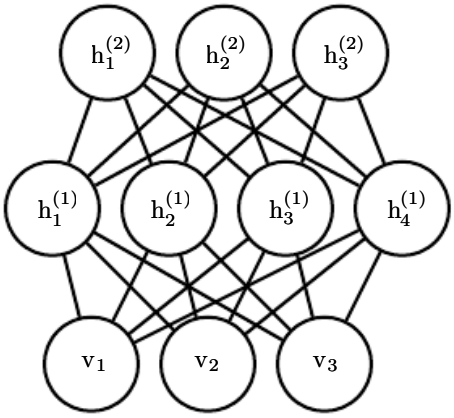

Figure 20.2: The graphical model for a deep Boltzmann machine with one visible layer (bottom) and two hidden layers. Connections are only between units in neighboring layers. There are no intralayer connections.

In comparison to the RBM energy function (equation 20.5), the DBM energy function includes connections between the hidden units (latent variables) in the form of the weight matrices ($\boldsymbol{W}^{(2)}$ and $\boldsymbol{W}^{(3)}$). As we will see, these connections have significant consequences for the model behavior as well as how we go about performing inference in the model.

In comparison to fully connected Boltzmann machines (with every unit connected to every other unit), the DBM offers some advantages that are similar to those offered by the RBM. Specifically, as illustrated in figure 20.3, the DBM layers can be organized into a bipartite graph, with odd layers on one side and even layers on the other. This immediately implies that when we condition on the variables in the even layer, the variables in the odd layers become conditionally independent. Of course, when we condition on the variables in the odd layers, the variables in the even layers also become conditionally independent.

The bipartite structure of the DBM means that we can apply the same equations we have previously used for the conditional distributions of an RBM to determine the conditional distributions in a DBM. The units within a layer are conditionally independent from each other given the values of the neighboring layers, so the distributions over binary variables can be fully described by the Bernoulli parameters, giving the probability of each unit being active. In our example with two hidden layers, the activation probabilities are given by

$$P(v_i = 1 \mid \boldsymbol{h}^{(1)}) = \sigma\left(\boldsymbol{W}^{(1)}_{i,:}\boldsymbol{h}^{(1)}\right), \tag{20.26}$$

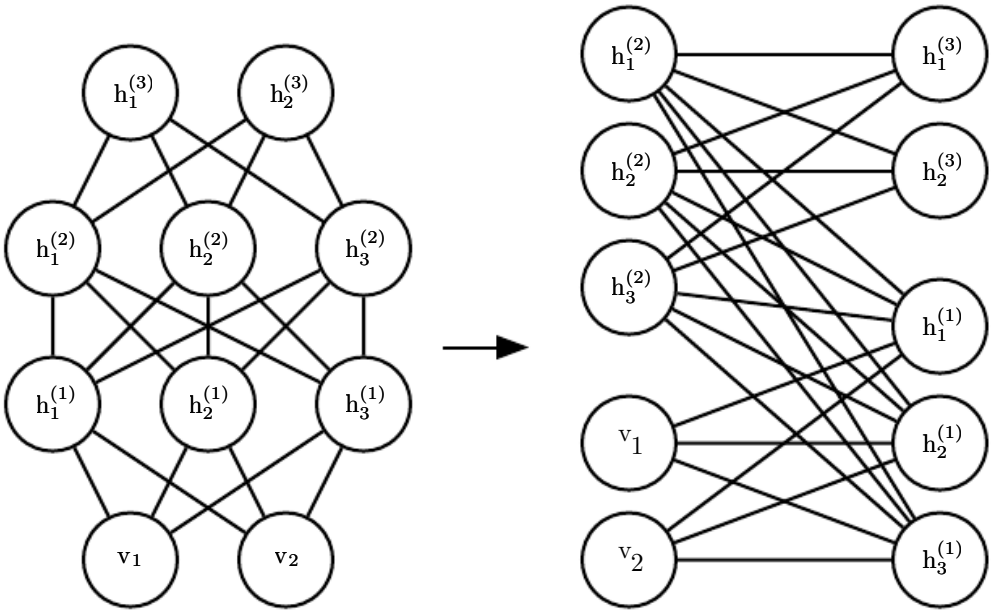

Figure 20.3: A deep Boltzmann machine, rearranged to reveal its bipartite graph structure.

$$P(h_i^{(1)} = 1 \mid \boldsymbol{v}, \boldsymbol{h}^{(2)}) = \sigma\left(\boldsymbol{v}^\top \boldsymbol{W}_{:,i}^{(1)} + \boldsymbol{W}_{i,:}^{(2)} \boldsymbol{h}^{(2)}\right), \tag{20.27}$$

and

$$P(h_k^{(2)} = 1 \mid \boldsymbol{h}^{(1)}) = \sigma\left(\boldsymbol{h}^{(1)\top} \boldsymbol{W}_{:,k}^{(2)}\right). \tag{20.28}$$

The bipartite structure makes Gibbs sampling in a deep Boltzmann machine efficient. The naive approach to Gibbs sampling is to update only one variable at a time. RBMs allow all the visible units to be updated in one block and all the hidden units to be updated in a second block. One might naively assume that a DBM with l layers requires $l + 1$ updates, with each iteration updating a block consisting of one layer of units. Instead, it is possible to update all the units in only two iterations. Gibbs sampling can be divided into two blocks of updates, one including all even layers (including the visible layer) and the other including all odd layers. Because of the bipartite DBM connection pattern, given the even layers, the distribution over the odd layers is factorial and thus can be sampled simultaneously and independently as a block. Likewise, given the odd layers, the even layers can be sampled simultaneously and independently as a block. Efficient sampling is especially important for training with the stochastic maximum likelihood algorithm.

20.4.1 Interesting Properties

Deep Boltzmann machines have many interesting properties.

DBMs were developed after DBNs. Compared to DBNs, the posterior distribution $P(\boldsymbol{h} \mid \boldsymbol{v})$ is simpler for DBMs. Somewhat counterintuitively, the simplicity of this posterior distribution allows richer approximations of the posterior. In the case of the DBN, we perform classification using a heuristically motivated approximate inference procedure, in which we guess that a reasonable value for the mean field expectation of the hidden units can be provided by an upward pass through the network in an MLP that uses sigmoid activation functions and the same weights as the original DBN. *Any* distribution $Q(\boldsymbol{h})$ can be used to obtain a variational lower bound on the log-likelihood. This heuristic procedure therefore enables us to obtain such a bound. Yet the bound is not explicitly optimized in any way, so it may be far from tight. In particular, the heuristic estimate of Q ignores interactions between hidden units within the same layer as well as the top-down feedback influence of hidden units in deeper layers on hidden units that are closer to the input. Because the heuristic MLP-based inference procedure in the DBN is not able to account for these interactions, the resulting Q is presumably

far from optimal. In DBMs, all the hidden units within a layer are conditionally independent given the other layers. This lack of intralayer interaction makes it possible to use fixed-point equations to optimize the variational lower bound and find the true optimal mean field expectations (to within some numerical tolerance).

The use of proper mean field allows the approximate inference procedure for DBMs to capture the influence of top-down feedback interactions. This makes DBMs interesting from the point of view of neuroscience, because the human brain is known to use many top-down feedback connections. Because of this property, DBMs have been used as computational models of real neuroscientific phenomena (Series *et al.*, 2010; Reichert *et al.*, 2011).

One unfortunate property of DBMs is that sampling from them is relatively difficult. DBNs only need to use MCMC sampling in their top pair of layers. The other layers are used only at the end of the sampling process, in one efficient ancestral sampling pass. To generate a sample from a DBM, it is necessary to use MCMC across all layers, with every layer of the model participating in every Markov chain transition.

20.4.2 DBM Mean Field Inference

The conditional distribution over one DBM layer given the neighboring layers is factorial. In the example of the DBM with two hidden layers, these distributions are $P(\boldsymbol{v} \mid \boldsymbol{h}^{(1)})$, $P(\boldsymbol{h}^{(1)} \mid \boldsymbol{v}, \boldsymbol{h}^{(2)})$, and $P(\boldsymbol{h}^{(2)} \mid \boldsymbol{h}^{(1)})$. The distribution over *all* hidden layers generally does not factorize because of interactions between layers. In the example with two hidden layers, $P(\boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)} \mid \boldsymbol{v})$ does not factorize because of the interaction weights $\boldsymbol{W}^{(2)}$ between $\boldsymbol{h}^{(1)}$ and $\boldsymbol{h}^{(2)}$, which render these variables mutually dependent.

As was the case with the DBN, we are left to seek out methods to approximate the DBM posterior distribution. Unlike the DBN, however, the DBM posterior distribution over their hidden units—while complicated—is easy to approximate with a variational approximation (as discussed in section 19.4), specifically a mean field approximation. The mean field approximation is a simple form of variational inference, where we restrict the approximating distribution to fully factorial distributions. In the context of DBMs, the mean field equations capture the bidirectional interactions between layers. In this section we derive the iterative approximate inference procedure originally introduced in Salakhutdinov and Hinton (2009a).

In variational approximations to inference, we approach the task of approximating a particular target distribution—in our case, the posterior distribution over

the hidden units given the visible units—by some reasonably simple family of distributions. In the case of the mean field approximation, the approximating family is the set of distributions where the hidden units are conditionally independent.

We now develop the mean field approach for the example with two hidden layers. Let $Q(\boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)} \mid \boldsymbol{v})$ be the approximation of $P(\boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)} \mid \boldsymbol{v})$. The mean field assumption implies that

$$Q(\boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)} \mid \boldsymbol{v}) = \prod_j Q(h_j^{(1)} \mid \boldsymbol{v}) \prod_k Q(h_k^{(2)} \mid \boldsymbol{v}). \tag{20.29}$$

The mean field approximation attempts to find a member of this family of distributions that best fits the true posterior $P(\boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)} \mid \boldsymbol{v})$. Importantly, the inference process must be run again to find a different distribution Q every time we use a new value of $\boldsymbol{v}$.

One can conceive of many ways of measuring how well $Q(\boldsymbol{h} \mid \boldsymbol{v})$ fits $P(\boldsymbol{h} \mid \boldsymbol{v})$. The mean field approach is to minimize

$$\mathrm{KL}(Q \| P) = \sum_{\boldsymbol{h}} Q(\boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)} \mid \boldsymbol{v}) \log \left(\frac{Q(\boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)} \mid \boldsymbol{v})}{P(\boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)} \mid \boldsymbol{v})} \right). \tag{20.30}$$

In general, we do not have to provide a parametric form of the approximating distribution beyond enforcing the independence assumptions. The variational approximation procedure is generally able to recover a functional form of the approximate distribution. However, in the case of a mean field assumption on binary hidden units (the case we are developing here) there is no loss of generality resulting from fixing a parametrization of the model in advance.

We parametrize Q as a product of Bernoulli distributions; that is, we associate the probability of each element of $\boldsymbol{h}^{(1)}$ with a parameter. Specifically, for each j, $\hat{h}_j^{(1)} = Q(h_j^{(1)} = 1 \mid \boldsymbol{v})$, where $\hat{h}_j^{(1)} \in [0, 1]$, and for each k, $\hat{h}_k^{(2)} = Q(h_k^{(2)} = 1 \mid \boldsymbol{v})$, where $\hat{h}_k^{(2)} \in [0, 1]$. Thus we have the following approximation to the posterior:

$$Q(\boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)} \mid \boldsymbol{v}) = \prod_j Q(h_j^{(1)} \mid \boldsymbol{v}) \prod_k Q(h_k^{(2)} \mid \boldsymbol{v}) \tag{20.31}$$

$$= \prod_j (\hat{h}_j^{(1)})^{h_j^{(1)}} (1 - \hat{h}_j^{(1)})^{(1 - h_j^{(1)})} \times \prod_k (\hat{h}_k^{(2)})^{h_k^{(2)}} (1 - \hat{h}_k^{(2)})^{(1 - h_k^{(2)})}. \tag{20.32}$$

Of course, for DBMs with more layers, the approximate posterior parametrization can be extended in the obvious way, exploiting the bipartite structure of the graph

to update all the even layers simultaneously and then to update all the odd layers simultaneously, following the same schedule as Gibbs sampling.

Now that we have specified our family of approximating distributions Q, it remains to specify a procedure for choosing the member of this family that best fits P. The most straightforward way to do this is to use the mean field equations specified by equation 19.56. These equations were derived by solving for where the derivatives of the variational lower bound are zero. They describe in an abstract manner how to optimize the variational lower bound for any model, simply by taking expectations with respect to Q.

Applying these general equations, we obtain the update rules (again, ignoring bias terms):

$$\hat{h}_j^{(1)} = \sigma\left(\sum_i v_i W_{i,j}^{(1)} + \sum_{k'} W_{j,k'}^{(2)} \hat{h}_{k'}^{(2)}\right), \quad \forall j, \tag{20.33}$$

$$\hat{h}_k^{(2)} = \sigma\left(\sum_{j'} W_{j',k}^{(2)} \hat{h}_{j'}^{(1)}\right), \quad \forall k. \tag{20.34}$$

At a fixed point of this system of equations, we have a local maximum of the variational lower bound $\mathcal{L}(Q)$. Thus these fixed-point update equations define an iterative algorithm where we alternate updates of $\hat{h}_j^{(1)}$ (using equation 20.33) and updates of $\hat{h}_k^{(2)}$ (using equation 20.34). On small problems such as MNIST, as few as ten iterations can be sufficient to find an approximate positive phase gradient for learning, and fifty usually suffice to obtain a high-quality representation of a single specific example to be used for high-accuracy classification. Extending approximate variational inference to deeper DBMs is straightforward.

20.4.3 DBM Parameter Learning

Learning in the DBM must confront both the challenge of an intractable partition function, using the techniques from chapter 18, and the challenge of an intractable posterior distribution, using the techniques from chapter 19.

As described in section 20.4.2, variational inference allows the construction of a distribution $Q(\boldsymbol{h} \mid \boldsymbol{v})$ that approximates the intractable $P(\boldsymbol{h} \mid \boldsymbol{v})$. Learning then proceeds by maximizing $\mathcal{L}(\boldsymbol{v}, Q, \boldsymbol{\theta})$, the variational lower bound on the intractable log-likelihood, $\log P(\boldsymbol{v}; \boldsymbol{\theta})$.

For a deep Boltzmann machine with two hidden layers, $\mathcal{L}$ is given by

$$\mathcal{L}(Q, \boldsymbol{\theta}) = \sum_i \sum_{j'} v_i W^{(1)}_{i,j'} \hat{h}^{(1)}_{j'} + \sum_{j'} \sum_{k'} \hat{h}^{(1)}_{j'} W^{(2)}_{j',k'} \hat{h}^{(2)}_{k'} - \log Z(\boldsymbol{\theta}) + \mathcal{H}(Q). \quad (20.35)$$

This expression still contains the log partition function, $\log Z(\boldsymbol{\theta})$. Because a deep Boltzmann machine contains restricted Boltzmann machines as components, the hardness results for computing the partition function and sampling that apply to restricted Boltzmann machines also apply to deep Boltzmann machines. This means that evaluating the probability mass function of a Boltzmann machine requires approximate methods such as annealed importance sampling. Likewise, training the model requires approximations to the gradient of the log partition function. See chapter 18 for a general description of these methods. DBMs are typically trained using stochastic maximum likelihood. Many of the other techniques described in chapter 18 are not applicable. Techniques such as pseudolikelihood require the ability to evaluate the unnormalized probabilities, rather than merely obtain a variational lower bound on them. Contrastive divergence is slow for deep Boltzmann machines because they do not allow efficient sampling of the hidden units given the visible units—instead, contrastive divergence would require burning in a Markov chain every time a new negative phase sample is needed.

The nonvariational version of the stochastic maximum likelihood algorithm is discussed in section 18.2. Variational stochastic maximum likelihood as applied to the DBM is given in algorithm 20.1. Recall that we describe a simplified variant of the DBM that lacks bias parameters; including them is trivial.

20.4.4 Layer-Wise Pretraining

Unfortunately, training a DBM using stochastic maximum likelihood (as described above) from a random initialization usually results in failure. In some cases, the model fails to learn to represent the distribution adequately. In other cases, the DBM may represent the distribution well, but with no higher likelihood than could be obtained with just an RBM. A DBM with very small weights in all but the first layer represents approximately the same distribution as an RBM.

Various techniques that permit joint training have been developed and are described in section 20.4.5. However, the original and most popular method for overcoming the joint training problem of DBMs is greedy layer-wise pretraining. In this method, each layer of the DBM is trained in isolation as an RBM. The first layer is trained to model the input data. Each subsequent RBM is trained to model samples from the previous RBM's posterior distribution. After all the

Algorithm 20.1 The variational stochastic maximum likelihood algorithm for training a DBM with two hidden layers

Set ϵ, the step size, to a small positive number
Set k, the number of Gibbs steps, high enough to allow a Markov chain of $p(\boldsymbol{v}, \boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)}; \boldsymbol{\theta} + \epsilon\Delta_{\boldsymbol{\theta}})$ to burn in, starting from samples from $p(\boldsymbol{v}, \boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)}; \boldsymbol{\theta})$.

Initialize three matrices, $\tilde{\boldsymbol{V}}$, $\tilde{\boldsymbol{H}}^{(1)}$, and $\tilde{\boldsymbol{H}}^{(2)}$ each with m rows set to random values (e.g., from Bernoulli distributions, possibly with marginals matched to the model's marginals).
while not converged (learning loop) **do**
  Sample a minibatch of m examples from the training data and arrange them as the rows of a design matrix $\boldsymbol{V}$.
  Initialize matrices $\hat{\boldsymbol{H}}^{(1)}$ and $\hat{\boldsymbol{H}}^{(2)}$, possibly to the model's marginals.
  while not converged (mean field inference loop) **do**
    $\hat{\boldsymbol{H}}^{(1)} \leftarrow \sigma\left(\boldsymbol{V}\boldsymbol{W}^{(1)} + \hat{\boldsymbol{H}}^{(2)}\boldsymbol{W}^{(2)\top}\right)$.
    $\hat{\boldsymbol{H}}^{(2)} \leftarrow \sigma\left(\hat{\boldsymbol{H}}^{(1)}\boldsymbol{W}^{(2)}\right)$.
  end while
  $\Delta_{\boldsymbol{W}^{(1)}} \leftarrow \frac{1}{m}\boldsymbol{V}^{\top}\hat{\boldsymbol{H}}^{(1)}$
  $\Delta_{\boldsymbol{W}^{(2)}} \leftarrow \frac{1}{m}\hat{\boldsymbol{H}}^{(1)\top}\hat{\boldsymbol{H}}^{(2)}$
  for $l = 1$ to k (Gibbs sampling) **do**
    Gibbs block 1:
    $\forall i, j, \tilde{V}_{i,j}$ sampled from $P(\tilde{V}_{i,j} = 1) = \sigma\left(\boldsymbol{W}^{(1)}_{j,:}\left(\tilde{\boldsymbol{H}}^{(1)}_{i,:}\right)^{\top}\right)$.
    $\forall i, j, \tilde{H}^{(2)}_{i,j}$ sampled from $P(\tilde{H}^{(2)}_{i,j} = 1) = \sigma\left(\tilde{\boldsymbol{H}}^{(1)}_{i,:}\boldsymbol{W}^{(2)}_{:,j}\right)$.
    Gibbs block 2:
    $\forall i, j, \tilde{H}^{(1)}_{i,j}$ sampled from $P(\tilde{H}^{(1)}_{i,j} = 1) = \sigma\left(\tilde{\boldsymbol{V}}_{i,:}\boldsymbol{W}^{(1)}_{:,j} + \tilde{\boldsymbol{H}}^{(2)}_{i,:}\boldsymbol{W}^{(2)\top}_{j,:}\right)$.
  end for
  $\Delta_{\boldsymbol{W}^{(1)}} \leftarrow \Delta_{\boldsymbol{W}^{(1)}} - \frac{1}{m}\boldsymbol{V}^{\top}\tilde{\boldsymbol{H}}^{(1)}$
  $\Delta_{\boldsymbol{W}^{(2)}} \leftarrow \Delta_{\boldsymbol{W}^{(2)}} - \frac{1}{m}\tilde{\boldsymbol{H}}^{(1)\top}\tilde{\boldsymbol{H}}^{(2)}$
  $\boldsymbol{W}^{(1)} \leftarrow \boldsymbol{W}^{(1)} + \epsilon\Delta_{\boldsymbol{W}^{(1)}}$ (this is a cartoon illustration, in practice use a more effective algorithm, such as momentum with a decaying learning rate)
  $\boldsymbol{W}^{(2)} \leftarrow \boldsymbol{W}^{(2)} + \epsilon\Delta_{\boldsymbol{W}^{(2)}}$
end while

RBMs have been trained in this way, they can be combined to form a DBM. The DBM may then be trained with PCD. Typically PCD training will make only a small change in the model's parameters and in its performance as measured by the

log-likelihood it assigns to the data, or its ability to classify inputs. See figure 20.4 for an illustration of the training procedure.

This greedy layer-wise training procedure is not just coordinate ascent. It bears some passing resemblance to coordinate ascent because we optimize one subset of the parameters at each step. The two methods differ because the greedy layer-wise training procedure uses a different objective function at each step.

Greedy layer-wise pretraining of a DBM differs from greedy layer-wise pretraining of a DBN. The parameters of each individual RBM may be copied to the corresponding DBN directly. In the case of the DBM, the RBM parameters must be modified before inclusion in the DBM. A layer in the middle of the stack of RBMs is trained with only bottom-up input, but after the stack is combined to form the DBM, the layer will have both bottom-up and top-down input. To account for this effect, Salakhutdinov and Hinton (2009a) advocate dividing the weights of all but the top and bottom RBM in half before inserting them into the DBM. Additionally, the bottom RBM must be trained using two "copies" of each visible unit and the weights tied to be equal between the two copies. This means that the weights are effectively doubled during the upward pass. Similarly, the top RBM should be trained with two copies of the topmost layer.

Obtaining the state of the art results with the deep Boltzmann machine requires a modification of the standard SML algorithm, which is to use a small amount of mean field during the negative phase of the joint PCD training step (Salakhutdinov and Hinton, 2009a). Specifically, the expectation of the energy gradient should be computed with respect to the mean field distribution in which all the units are independent from each other. The parameters of this mean field distribution should be obtained by running the mean field fixed-point equations for just one step. See Goodfellow *et al.* (2013b) for a comparison of the performance of centered DBMs with and without the use of partial mean field in the negative phase.

20.4.5 Jointly Training Deep Boltzmann Machines

Classic DBMs require greedy unsupervised pretraining and, to perform classification well, require a separate MLP-based classifier on top of the hidden features they extract. This has some undesirable properties. It is hard to track performance during training because we cannot evaluate properties of the full DBM while training the first RBM. Thus, it is hard to tell how well our hyperparameters are working until quite late in the training process. Software implementations of DBMs need to have many different components for CD training of individual RBMs, PCD training of the full DBM, and training based on back-propagation

through the MLP. Finally, the MLP on top of the Boltzmann machine loses many of the advantages of the Boltzmann machine probabilistic model, such as being able to perform inference when some input values are missing.

There are two main ways to resolve the joint training problem of the deep

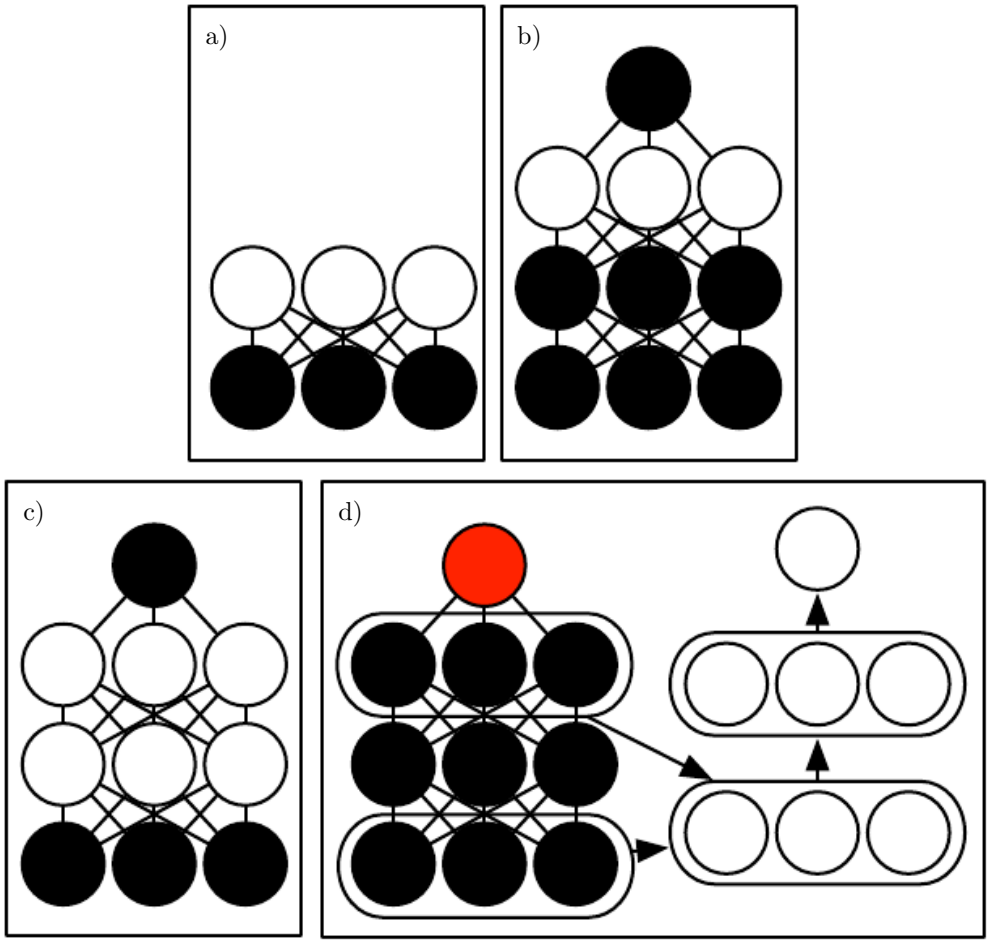

Figure 20.4: The deep Boltzmann machine training procedure used to classify the MNIST dataset (Salakhutdinov and Hinton, 2009a; Srivastava *et al.*, 2014). *(a)*Train an RBM by using CD to approximately maximize $\log P(\boldsymbol{v})$. *(b)*Train a second RBM that models $\boldsymbol{h}^{(1)}$ and target class y by using CD-k to approximately maximize $\log P(\boldsymbol{h}^{(1)}, \mathrm{y})$, where $\boldsymbol{h}^{(1)}$ is drawn from the first RBM's posterior conditioned on the data. Increase k from 1 to 20 during learning. *(c)*Combine the two RBMs into a DBM. Train it to approximately maximize $\log P(\mathbf{v}, \mathrm{y})$ using stochastic maximum likelihood with $k = 5$. *(d)*Delete y from the model. Define a new set of features $\boldsymbol{h}^{(1)}$ and $\boldsymbol{h}^{(2)}$ that are obtained by running mean field inference in the model lacking y. Use these features as input to an MLP whose structure is the same as an additional pass of mean field, with an additional output layer for the estimate of y. Initialize the MLP's weights to be the same as the DBM's weights. Train the MLP to approximately maximize $\log P(\mathrm{y} \mid \mathbf{v})$ using stochastic gradient descent and dropout. Figure reprinted from Goodfellow *et al.* (2013b).

Boltzmann machine. The first is the **centered deep Boltzmann machine** (Montavon and Muller, 2012), which reparametrizes the model in order to make the Hessian of the cost function better conditioned at the beginning of the learning process. This yields a model that can be trained without a greedy layer-wise pretraining stage. The resulting model obtains excellent test set log-likelihood and produces high-quality samples. Unfortunately, it remains unable to compete with appropriately regularized MLPs as a classifier. The second way to jointly train a deep Boltzmann machine is to use a **multi-prediction deep Boltzmann machine** (Goodfellow *et al.*, 2013b). This model uses an alternative training criterion that allows the use of the back-propagation algorithm to avoid the problems with MCMC estimates of the gradient. Unfortunately, the new criterion does not lead to good likelihood or samples, but, compared to the MCMC approach, it does lead to superior classification performance and ability to reason well about missing inputs.

The centering trick for the Boltzmann machine is easiest to describe if we return to the general view of a Boltzmann machine as consisting of a set of units $\boldsymbol{x}$ with a weight matrix $\boldsymbol{U}$ and biases $\boldsymbol{b}$. Recall from equation 20.2 that the energy function is given by

$$E(\boldsymbol{x}) = -\boldsymbol{x}^\top \boldsymbol{U}\boldsymbol{x} - \boldsymbol{b}^\top \boldsymbol{x}. \tag{20.36}$$

Using different sparsity patterns in the weight matrix $\boldsymbol{U}$, we can implement structures of Boltzmann machines, such as RBMs or DBMs with different numbers of layers. This is accomplished by partitioning $\boldsymbol{x}$ into visible and hidden units and zeroing out elements of $\boldsymbol{U}$ for units that do not interact. The centered Boltzmann machine introduces a vector $\boldsymbol{\mu}$ that is subtracted from all the states:

$$E'(\boldsymbol{x}; \boldsymbol{U}, \boldsymbol{b}) = -(\boldsymbol{x} - \boldsymbol{\mu})^\top \boldsymbol{U}(\boldsymbol{x} - \boldsymbol{\mu}) - (\boldsymbol{x} - \boldsymbol{\mu})^\top \boldsymbol{b}. \tag{20.37}$$

Typically $\boldsymbol{\mu}$ is a hyperparameter fixed at the beginning of training. It is usually chosen to make sure that $\boldsymbol{x} - \boldsymbol{\mu} \approx \boldsymbol{0}$ when the model is initialized. This reparametrization does not change the set of probability distributions that the model can represent, but it does change the dynamics of stochastic gradient descent applied to the likelihood. Specifically, in many cases, this reparametrization results in a Hessian matrix that is better conditioned. Melchior *et al.* (2013) experimentally confirmed that the conditioning of the Hessian matrix improves, and observed that the centering trick is equivalent to another Boltzmann machine learning technique, the **enhanced gradient** (Cho *et al.*, 2011). The improved conditioning of the Hessian matrix enables learning to succeed, even in difficult cases like training a deep Boltzmann machine with multiple layers.

The other approach to jointly training deep Boltzmann machines is the multi-prediction deep Boltzmann machine (MP-DBM), which works by viewing the mean

field equations as defining a family of recurrent networks for approximately solving every possible inference problem (Goodfellow *et al.*, 2013b). Rather than training the model to maximize the likelihood, the model is trained to make each recurrent network obtain an accurate answer to the corresponding inference problem. The training process is illustrated in figure 20.5. It consists of randomly sampling a training example, randomly sampling a subset of inputs to the inference network, and then training the inference network to predict the values of the remaining units.

This general principle of back-propagating through the computational graph for approximate inference has been applied to other models (Stoyanov *et al.*, 2011; Brakel *et al.*, 2013). In these models and in the MP-DBM, the final loss is not the lower bound on the likelihood. Instead, the final loss is typically based on the approximate conditional distribution that the approximate inference network imposes over the missing values. This means that the training of these models is somewhat heuristically motivated. If we inspect the $p(\boldsymbol{v})$ represented by the Boltzmann machine learned by the MP-DBM, it tends to be somewhat defective, in the sense that Gibbs sampling yields poor samples.

Back-propagation through the inference graph has two main advantages. First, it trains the model as it is really used—with approximate inference. This means that approximate inference, for example, to fill in missing inputs or to perform classification despite the presence of missing inputs, is more accurate in the MP-DBM than in the original DBM. The original DBM does not make an accurate classifier on its own; the best classification results with the original DBM were based on training a separate classifier to use features extracted by the DBM, rather than by using inference in the DBM to compute the distribution over the class labels. Mean field inference in the MP-DBM performs well as a classifier without special modifications. The other advantage of back-propagating through approximate inference is that back-propagation computes the exact gradient of the loss. This is better for optimization than the approximate gradients of SML training, which suffer from both bias and variance. This probably explains why MP-DBMs may be trained jointly while DBMs require a greedy layer-wise pretraining. The disadvantage of back-propagating through the approximate inference graph is that it does not provide a way to optimize the log-likelihood, but rather gives a heuristic approximation of the generalized pseudolikelihood.

The MP-DBM inspired the NADE-k (Raiko *et al.*, 2014) extension to the NADE framework, which is described in section 20.10.10.

The MP-DBM has some connections to dropout. Dropout shares the same parameters among many different computational graphs, with the difference between

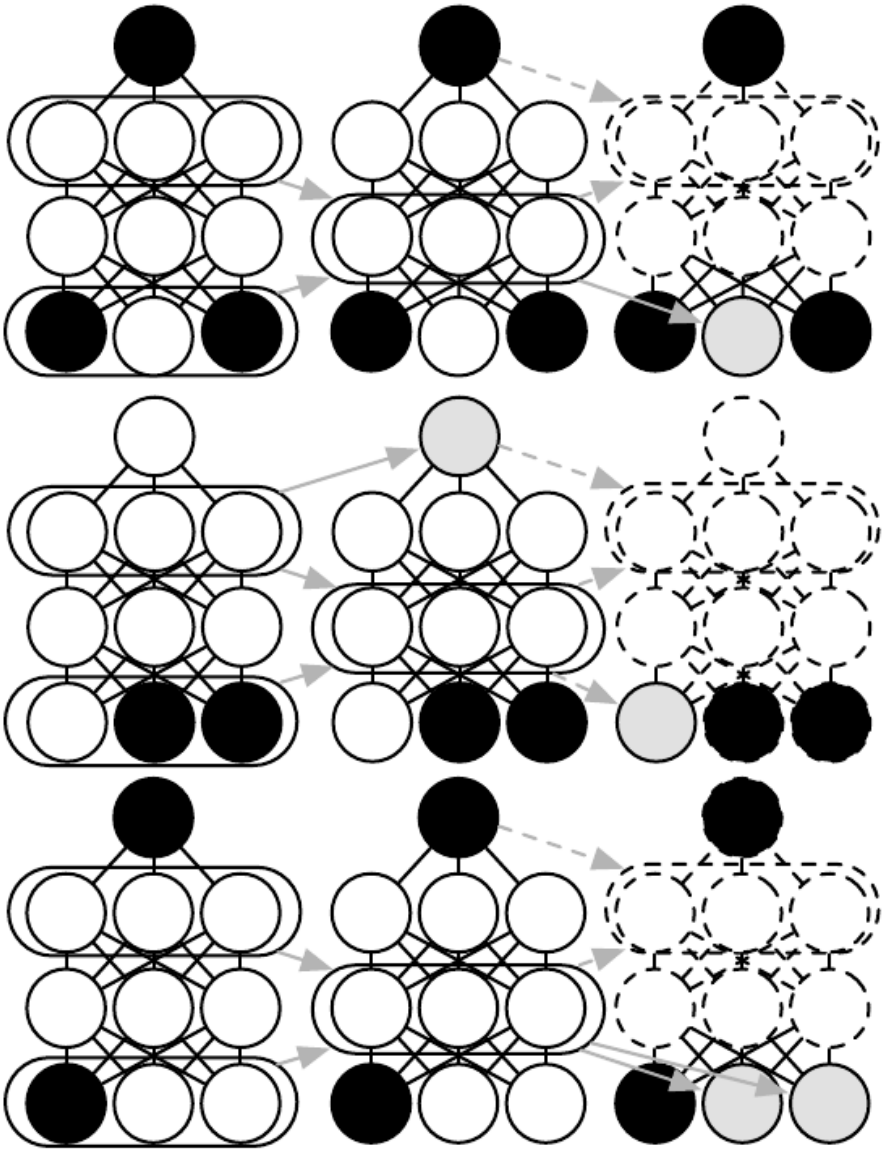

Figure 20.5: An illustration of the multiprediction training process for a deep Boltzmann machine. Each row indicates a different example within a minibatch for the same training step. Each column represents a time step within the mean field inference process. For each example, we sample a subset of the data variables to serve as inputs to the inference process. These variables are shaded black to indicate conditioning. We then run the mean field inference process, with arrows indicating which variables influence which other variables in the process. In practical applications, we unroll mean field for several steps. In this illustration, we unroll for only two steps. Dashed arrows indicate how the process could be unrolled for more steps. The data variables that were not used as inputs to the inference process become targets, shaded in gray. We can view the inference process for each example as a recurrent network. We use gradient descent and back-propagation to train these recurrent networks to produce the correct targets given their inputs. This trains the mean field process for the MP-DBM to produce accurate estimates. Figure adapted from Goodfellow *et al.* (2013b).

each graph being whether it includes or excludes each unit. The MP-DBM also shares parameters across many computational graphs. In the case of the MP-DBM, the difference between the graphs is whether each input unit is observed or not. When a unit is not observed, the MP-DBM does not delete it entirely as dropout does. Instead, the MP-DBM treats it as a latent variable to be inferred. One could imagine applying dropout to the MP-DBM by additionally removing some units rather than making them latent.

20.5 Boltzmann Machines for Real-Valued Data

While Boltzmann machines were originally developed for use with binary data, many applications such as image and audio modeling seem to require the ability to represent probability distributions over real values. In some cases, it is possible to treat real-valued data in the interval [0, 1] as representing the expectation of a binary variable. For example, Hinton (2000) treats grayscale images in the training set as defining [0,1] probability values. Each pixel defines the probability of a binary value being 1, and the binary pixels are all sampled independently from each other. This is a common procedure for evaluating binary models on grayscale image datasets. Nonetheless, it is not a particularly theoretically satisfying approach, and binary images sampled independently in this way have a noisy appearance. In this section, we present Boltzmann machines that define a probability density over real-valued data.

20.5.1 Gaussian-Bernoulli RBMs

Restricted Boltzmann machines may be developed for many exponential family conditional distributions (Welling *et al.*, 2005). Of these, the most common is the RBM with binary hidden units and real-valued visible units, with the conditional distribution over the visible units being a Gaussian distribution whose mean is a function of the hidden units.

There are many ways of parametrizing Gaussian-Bernoulli RBMs. One choice is whether to use a covariance matrix or a precision matrix for the Gaussian distribution. Here we present the precision formulation. The modification to obtain the covariance formulation is straightforward. We wish to have the conditional distribution

$$p(\boldsymbol{v} \mid \boldsymbol{h}) = \mathcal{N}(\boldsymbol{v}; \boldsymbol{W}\boldsymbol{h}, \boldsymbol{\beta}^{-1}). \tag{20.38}$$

We can find the terms we need to add to the energy function by expanding the

unnormalized log conditional distribution:

$$\log \mathcal{N}(\boldsymbol{v}; \boldsymbol{W}\boldsymbol{h}, \boldsymbol{\beta}^{-1}) = -\frac{1}{2}(\boldsymbol{v} - \boldsymbol{W}\boldsymbol{h})^\top \boldsymbol{\beta}(\boldsymbol{v} - \boldsymbol{W}\boldsymbol{h}) + f(\boldsymbol{\beta}). \tag{20.39}$$

Here f encapsulates all the terms that are a function only of the parameters and not the random variables in the model. We can discard f because its only role is to normalize the distribution, and the partition function of whatever energy function we choose will carry out that role.

If we include all the terms (with their sign flipped) involving $\boldsymbol{v}$ from equation 20.39 in our energy function and do not add any other terms involving $\boldsymbol{v}$, then our energy function will represent the desired conditional $p(\boldsymbol{v} \mid \boldsymbol{h})$.

We have some freedom regarding the other conditional distribution, $p(\boldsymbol{h} \mid \boldsymbol{v})$. Note that equation 20.39 contains a term

$$\frac{1}{2}\boldsymbol{h}^\top \boldsymbol{W}^\top \boldsymbol{\beta} \boldsymbol{W} \boldsymbol{h}. \tag{20.40}$$

This term cannot be included in its entirety because it includes $h_i h_j$ terms. These correspond to edges between the hidden units. If we included these terms, we would have a linear factor model instead of a restricted Boltzmann machine. When designing our Boltzmann machine, we simply omit these $h_i h_j$ cross terms. Omitting them does not change the conditional $p(\boldsymbol{v} \mid \boldsymbol{h})$, so equation 20.39 is still respected. We still have a choice, however, about whether to include the terms involving only a single h_i. If we assume a diagonal precision matrix, we find that for each hidden unit h_i, we have a term

$$\frac{1}{2} h_i \sum_j \beta_j W_{j,i}^2. \tag{20.41}$$

In the above, we used the fact that $h_i^2 = h_i$ because $h_i \in \{0, 1\}$. If we include this term (with its sign flipped) in the energy function, then it will naturally bias h_i to be turned off when the weights for that unit are large and connected to visible units with high precision. The choice of whether to include this bias term does not affect the family of distributions that the model can represent (assuming that we include bias parameters for the hidden units), but it does affect the learning dynamics of the model. Including the term may help the hidden unit activations remain reasonable even when the weights rapidly increase in magnitude.

One way to define the energy function on a Gaussian-Bernoulli RBM is thus:

$$E(\boldsymbol{v}, \boldsymbol{h}) = \frac{1}{2}\boldsymbol{v}^\top(\boldsymbol{\beta} \odot \boldsymbol{v}) - (\boldsymbol{v} \odot \boldsymbol{\beta})^\top \boldsymbol{W}\boldsymbol{h} - \boldsymbol{b}^\top \boldsymbol{h}, \tag{20.42}$$

but we may also add extra terms or parametrize the energy in terms of the variance rather than precision if we choose.

In this derivation, we have not included a bias term on the visible units, but one could easily be added. One final source of variability in the parametrization of a Gaussian-Bernoulli RBM is the choice of how to treat the precision matrix. It may be either fixed to a constant (perhaps estimated based on the marginal precision of the data) or learned. It may also be a scalar times the identity matrix, or it may be a diagonal matrix. Typically we do not allow the precision matrix to be nondiagonal in this context, because some operations on the Gaussian distribution require inverting the matrix, and a diagonal matrix can be inverted trivially. In the sections ahead, we will see that other forms of Boltzmann machines permit modeling the covariance structure, using various techniques to avoid inverting the precision matrix.

20.5.2 Undirected Models of Conditional Covariance

While the Gaussian RBM has been the canonical energy model for real-valued data, Ranzato *et al.* (2010a) argue that the Gaussian RBM inductive bias is not well suited to the statistical variations present in some types of real-valued data, especially natural images. The problem is that much of the information content present in natural images is embedded in the covariance between pixels rather than in the raw pixel values. In other words, it is the relationships between pixels and not their absolute values where most of the useful information in images resides. Since the Gaussian RBM only models the conditional mean of the input given the hidden units, it cannot capture conditional covariance information. In response to these criticisms, alternative models have been proposed that attempt to better account for the covariance of real-valued data. These models include the mean and covariance RBM (mcRBM[1]), the mean product of Student t-distribution (mPoT) model, and the spike and slab RBM (ssRBM).

Mean and Covariance RBM The mcRBM uses its hidden units to independently encode the conditional mean and covariance of all observed units. The mcRBM hidden layer is divided into two groups of units: mean units and covariance units. The group that models the conditional mean is simply a Gaussian RBM. The other half is a covariance RBM (Ranzato *et al.*, 2010a), also called a cRBM, whose components model the conditional covariance structure, as described below.

[1]The term "mcRBM" is pronounced by saying the name of the letters M-C-R-B-M; the "mc" is not pronounced like the "Mc" in "McDonald's."

Specifically, with binary mean units $\boldsymbol{h}^{(m)}$ and binary covariance units $\boldsymbol{h}^{(c)}$, the mcRBM model is defined as the combination of two energy functions:

$$E_{\text{mc}}\left(\boldsymbol{x}, \boldsymbol{h}^{(m)}, \boldsymbol{h}^{(c)}\right) = E_{\text{m}}(\boldsymbol{x}, \boldsymbol{h}^{(m)}) + E_{\text{c}}(\boldsymbol{x}, \boldsymbol{h}^{(c)}), \tag{20.43}$$

where E_{m} is the standard Gaussian-Bernoulli RBM energy function,[2]

$$E_{\text{m}}(\boldsymbol{x}, \boldsymbol{h}^{(m)}) = \frac{1}{2}\boldsymbol{x}^{\top}\boldsymbol{x} - \sum_j \boldsymbol{x}^{\top}\boldsymbol{W}_{:,j}h_j^{(m)} - \sum_j b_j^{(m)}h_j^{(m)}, \tag{20.44}$$

and E_{c} is the cRBM energy function that models the conditional covariance information:

$$E_{\text{c}}\left(\boldsymbol{x}, \boldsymbol{h}^{(c)}\right) = \frac{1}{2}\sum_j h_j^{(c)}\left(\boldsymbol{x}^{\top}\boldsymbol{r}^{(j)}\right)^2 - \sum_j b_j^{(c)}h_j^{(c)}. \tag{20.45}$$

The parameter $\boldsymbol{r}^{(j)}$ corresponds to the covariance weight vector associated with $h_j^{(c)}$, and $\boldsymbol{b}^{(c)}$ is a vector of covariance offsets. The combined energy function defines a joint distribution,

$$p_{\text{mc}}(\boldsymbol{x}, \boldsymbol{h}^{(m)}, \boldsymbol{h}^{(c)}) = \frac{1}{Z}\exp\Big\{-E_{\text{mc}}(\boldsymbol{x}, \boldsymbol{h}^{(m)}, \boldsymbol{h}^{(c)})\Big\}, \tag{20.46}$$

and a corresponding conditional distribution over the observations given $\boldsymbol{h}^{(m)}$ and $\boldsymbol{h}^{(c)}$ as a multivariate Gaussian distribution:

$$p_{\text{mc}}(\boldsymbol{x} \mid \boldsymbol{h}^{(m)}, h^{(c)}) = \mathcal{N}\left(\boldsymbol{x}; \boldsymbol{C}_{\boldsymbol{x}|\boldsymbol{h}}^{\text{mc}}\left(\sum_j \boldsymbol{W}_{:,j}h_j^{(m)}\right), \boldsymbol{C}_{\boldsymbol{x}|\boldsymbol{h}}^{\text{mc}}\right). \tag{20.47}$$

Note that the covariance matrix $\boldsymbol{C}_{\boldsymbol{x}|\boldsymbol{h}}^{\text{mc}} = \left(\sum_j h_j^{(c)}\boldsymbol{r}^{(j)}\boldsymbol{r}^{(j)\top} + \boldsymbol{I}\right)^{-1}$ is nondiagonal and that $\boldsymbol{W}$ is the weight matrix associated with the Gaussian RBM modeling the conditional means. It is difficult to train the mcRBM via contrastive divergence or persistent contrastive divergence because of its nondiagonal conditional covariance structure. CD and PCD require sampling from the joint distribution of $\boldsymbol{x}, \boldsymbol{h}^{(m)}, \boldsymbol{h}^{(c)}$, which, in a standard RBM, is accomplished by Gibbs sampling over the conditionals. However, in the mcRBM, sampling from $p_{\text{mc}}(\boldsymbol{x} \mid \boldsymbol{h}^{(m)}, \boldsymbol{h}^{(c)})$ requires computing $(\boldsymbol{C}^{\text{mc}})^{-1}$ at every iteration of learning. This can be an impractical computational burden for larger observations. Ranzato and Hinton (2010) avoid direct sampling from the conditional $p_{\text{mc}}(\boldsymbol{x} \mid \boldsymbol{h}^{(m)}, \boldsymbol{h}^{(c)})$ by sampling directly from the marginal $p(\boldsymbol{x})$ using Hamiltonian (hybrid) Monte Carlo (Neal, 1993) on the mcRBM free energy.

[2]This version of the Gaussian-Bernoulli RBM energy function assumes the image data have zero mean per pixel. Pixel offsets can easily be added to the model to account for nonzero pixel means.

Mean Product of Student t-distributions The mean product of Student t-distribution (mPoT) model (Ranzato *et al.*, 2010b) extends the PoT model (Welling *et al.*, 2003a) in a manner similar to how the mcRBM extends the cRBM. This is achieved by including nonzero Gaussian means by the addition of Gaussian RBM-like hidden units. Like the mcRBM, the PoT conditional distribution over the observation is a multivariate Gaussian (with nondiagonal covariance) distribution; however, unlike the mcRBM, the complementary conditional distribution over the hidden variables is given by conditionally independent Gamma distributions. The Gamma distribution $\mathcal{G}(k, \theta)$ is a probability distribution over positive real numbers, with mean $k\theta$. It is not necessary to have a more detailed understanding of the Gamma distribution to understand the basic ideas underlying the mPoT model.

The mPoT energy function is

$$E_{\text{mPoT}}(\boldsymbol{x}, \boldsymbol{h}^{(m)}, \boldsymbol{h}^{(c)}) \tag{20.48}$$

$$= E_m(\boldsymbol{x}, \boldsymbol{h}^{(m)}) + \sum_j \left(h_j^{(c)} \left(1 + \frac{1}{2} \left(\boldsymbol{r}^{(j)\top} \boldsymbol{x} \right)^2 \right) + (1 - \gamma_j) \log h_j^{(c)} \right), \tag{20.49}$$

where $\boldsymbol{r}^{(j)}$ is the covariance weight vector associated with unit $h_j^{(c)}$, and $E_m(\boldsymbol{x}, \boldsymbol{h}^{(m)})$ is as defined in equation 20.44.

Just as with the mcRBM, the mPoT model energy function specifies a multivariate Gaussian, with a conditional distribution over $\boldsymbol{x}$ that has nondiagonal covariance. Learning in the mPoT model—again, like the mcRBM—is complicated by the inability to sample from the nondiagonal Gaussian conditional $p_{\text{mPoT}}(\boldsymbol{x} \mid \boldsymbol{h}^{(m)}, \boldsymbol{h}^{(c)})$, so Ranzato *et al.* (2010b) also advocate direct sampling of $p(\boldsymbol{x})$ via Hamiltonian (hybrid) Monte Carlo.

Spike and Slab Restricted Boltzmann Machines Spike and slab restricted Boltzmann machines (Courville *et al.*, 2011) or ssRBMs provide another means of modeling the covariance structure of real-valued data. Compared to mcRBMs, ssRBMs have the advantage of requiring neither matrix inversion nor Hamiltonian Monte Carlo methods. Like the mcRBM and the mPoT model, the ssRBM's binary hidden units encode the conditional covariance across pixels through the use of auxiliary real-valued variables.

The spike and slab RBM has two sets of hidden units: binary **spike** units $\mathbf{h}$ and real-valued **slab** units $\mathbf{s}$. The mean of the visible units conditioned on the hidden units is given by $(\boldsymbol{h} \odot \boldsymbol{s})\boldsymbol{W}^\top$. In other words, each column $\boldsymbol{W}_{:,i}$ defines a component that can appear in the input when $h_i = 1$. The corresponding spike

variable h_i determines whether that component is present at all. The corresponding slab variable s_i determines the intensity of that component, if it is present. When a spike variable is active, the corresponding slab variable adds variance to the input along the axis defined by $\boldsymbol{W}_{:,i}$. This allows us to model the covariance of the inputs. Fortunately, contrastive divergence and persistent contrastive divergence with Gibbs sampling are still applicable. There is no need to invert any matrix.

Formally, the ssRBM model is defined via its energy function:

$$E_{\mathrm{ss}}(\boldsymbol{x}, \boldsymbol{s}, \boldsymbol{h}) = -\sum_i \boldsymbol{x}^\top \boldsymbol{W}_{:,i} s_i h_i \; + \frac{1}{2}\boldsymbol{x}^\top \left(\boldsymbol{\Lambda} + \sum_i \boldsymbol{\Phi}_i h_i\right)\boldsymbol{x} \tag{20.50}$$

$$+ \frac{1}{2}\sum_i \alpha_i s_i^2 \; - \sum_i \alpha_i \mu_i s_i h_i \; - \sum_i b_i h_i + \sum_i \alpha_i \mu_i^2 h_i, \tag{20.51}$$

where b_i is the offset of the spike h_i, and $\boldsymbol{\Lambda}$ is a diagonal precision matrix on the observations $\boldsymbol{x}$. The parameter $\alpha_i > 0$ is a scalar precision parameter for the real-valued slab variable $\boldsymbol{s}_i$. The parameter $\boldsymbol{\Phi}_i$ is a nonnegative diagonal matrix that defines an $\boldsymbol{h}$-modulated quadratic penalty on $\boldsymbol{x}$. Each μ_i is a mean parameter for the slab variable s_i.

With the joint distribution defined via the energy function, deriving the ssRBM conditional distributions is relatively straightforward. For example, by marginalizing out the slab variables $\boldsymbol{s}$, the conditional distribution over the observations given the binary spike variables $\boldsymbol{h}$ is given by

$$p_{\mathrm{ss}}(\boldsymbol{x} \mid \boldsymbol{h}) \;=\; \frac{1}{P(\boldsymbol{h})}\frac{1}{Z}\int \exp\left\{-E(\boldsymbol{x}, \boldsymbol{s}, \boldsymbol{h})\right\}\; d\boldsymbol{s} \tag{20.52}$$

$$=\; \mathcal{N}\left(\boldsymbol{x}; \boldsymbol{C}^{\mathrm{ss}}_{\boldsymbol{x}|\boldsymbol{h}} \sum_i \boldsymbol{W}_{:,i}\mu_i h_i \;,\; \boldsymbol{C}^{\mathrm{ss}}_{\boldsymbol{x}|\boldsymbol{h}}\right) \tag{20.53}$$

where $\boldsymbol{C}^{\mathrm{ss}}_{\boldsymbol{x}|\boldsymbol{h}} = \left(\boldsymbol{\Lambda} + \sum_i \boldsymbol{\Phi}_i h_i - \sum_i \alpha_i^{-1} h_i \boldsymbol{W}_{:,i}\boldsymbol{W}_{:,i}^\top\right)^{-1}$. The last equality holds only if the covariance matrix $\boldsymbol{C}^{\mathrm{ss}}_{\boldsymbol{x}|\boldsymbol{h}}$ is positive definite.

Gating by the spike variables means that the true marginal distribution over $\mathbf{h} \odot \mathbf{s}$ is sparse. This is different from sparse coding, where samples from the model "almost never" (in the measure theoretic sense) contain zeros in the code, and MAP inference is required to impose sparsity.

Comparing the ssRBM to the mcRBM and the mPoT models, the ssRBM parametrizes the conditional covariance of the observation in a significantly different way. The mcRBM and mPoT both model the covariance structure of the observation as $\left(\sum_j h_j^{(c)} \boldsymbol{r}^{(j)} \boldsymbol{r}^{(j)\top} + \boldsymbol{I}\right)^{-1}$, using the activation of the hidden units $\boldsymbol{h}_j > 0$ to

enforce constraints on the conditional covariance in the direction $\boldsymbol{r}^{(j)}$. In contrast, the ssRBM specifies the conditional covariance of the observations using the hidden spike activations $h_i = 1$ to pinch the precision matrix along the direction specified by the corresponding weight vector. The ssRBM conditional covariance is similar to that given by a different model: the product of probabilistic principal components analysis (PoPPCA) (Williams and Agakov, 2002). In the overcomplete setting, sparse activations with the ssRBM parametrization permit significant variance (above the nominal variance given by $\boldsymbol{\Lambda}^{-1}$) only in the selected directions of the sparsely activated h_i. In the mcRBM or mPoT models, an overcomplete representation would mean that to capture variation in a particular direction in the observation space would require removing potentially all constraints with positive projection in that direction. This would suggest that these models are less well suited to the overcomplete setting.

The primary disadvantage of the spike and slab restricted Boltzmann machine is that some settings of the parameters can correspond to a covariance matrix that is not positive definite. Such a covariance matrix places more unnormalized probability on values that are farther from the mean, causing the integral over all possible outcomes to diverge. Generally this issue can be avoided with simple heuristic tricks. There is not yet any theoretically satisfying solution. Using constrained optimization to explicitly avoid the regions where the probability is undefined is difficult to do without being overly conservative and also preventing the model from accessing high-performing regions of parameter space.

Qualitatively, convolutional variants of the ssRBM produce excellent samples of natural images. Some examples are shown in figure 16.1.

The ssRBM allows for several extensions. Including higher-order interactions and average-pooling of the slab variables (Courville *et al.*, 2014) enables the model to learn excellent features for a classifier when labeled data is scarce. Adding a term to the energy function that prevents the partition function from becoming undefined results in a sparse coding model, spike and slab sparse coding (Goodfellow *et al.*, 2013d), also known as S3C.

20.6 Convolutional Boltzmann Machines

As we discuss in chapter 9, extremely high-dimensional inputs such as images place great strain on the computation, memory and statistical requirements of machine learning models. Replacing matrix multiplication by discrete convolution with a small kernel is the standard way of solving these problems for inputs that have translation invariant spatial or temporal structure. Desjardins and Bengio (2008)

showed that this approach works well when applied to RBMs.

Deep convolutional networks usually require a pooling operation so that the spatial size of each successive layer decreases. Feedforward convolutional networks often use a pooling function such as the maximum of the elements to be pooled. It is unclear how to generalize this to the setting of energy-based models. We could introduce a binary pooling unit p over n binary detector units $\mathbf{d}$ and enforce $p = \max_i d_i$ by setting the energy function to be ∞ whenever that constraint is violated. This does not scale well though, as it requires evaluating 2^n different energy configurations to compute the normalization constant. For a small 3×3 pooling region this requires $2^9 = 512$ energy function evaluations per pooling unit!

Lee *et al.* (2009) developed a solution to this problem called **probabilistic max pooling** (not to be confused with "stochastic pooling," which is a technique for implicitly constructing ensembles of convolutional feedforward networks). The strategy behind probabilistic max pooling is to constrain the detector units so at most one may be active at a time. This means there are only $n + 1$ total states (one state for each of the n detector units being on, and an additional state corresponding to all the detector units being off). The pooling unit is on if and only if one of the detector units is on. The state with all units off is assigned energy zero. We can think of this as describing a model with a single variable that has $n + 1$ states, or equivalently as a model that has $n + 1$ variables that assigns energy ∞ to all but $n + 1$ joint assignments of variables.

While efficient, probabilistic max pooling does force the detector units to be mutually exclusive, which may be a useful regularizing constraint in some contexts or a harmful limit on model capacity in other contexts. It also does not support overlapping pooling regions. Overlapping pooling regions are usually required to obtain the best performance from feedforward convolutional networks, so this constraint probably greatly reduces the performance of convolutional Boltzmann machines.

Lee *et al.* (2009) demonstrated that probabilistic max pooling could be used to build convolutional deep Boltzmann machines.[3] This model is able to perform operations such as filling in missing portions of its input. While intellectually appealing, this model is challenging to make work in practice, and usually does not perform as well as a classifier as traditional convolutional networks trained with supervised learning.

Many convolutional models work equally well with inputs of many different

[3]The publication describes the model as a "deep belief network," but because it can be described as a purely undirected model with tractable layer-wise mean field fixed-point updates, it best fits the definition of a deep Boltzmann machine.

spatial sizes. For Boltzmann machines, it is difficult to change the input size for various reasons. The partition function changes as the size of the input changes. Moreover, many convolutional networks achieve size invariance by scaling up the size of their pooling regions proportional to the size of the input, but scaling Boltzmann machine pooling regions is awkward. Traditional convolutional neural networks can use a fixed number of pooling units and dynamically increase the size of their pooling regions to obtain a fixed-size representation of a variably sized input. For Boltzmann machines, large pooling regions become too expensive for the naive approach. The approach of Lee *et al.* (2009) of making each of the detector units in the same pooling region mutually exclusive solves the computational problems but still does not allow variably sized pooling regions. For example, suppose we learn a model with 2×2 probabilistic max pooling over detector units that learn edge detectors. This enforces the constraint that only one of these edges may appear in each 2×2 region. If we then increase the size of the input image by 50 percent in each direction, we would expect the number of edges to increase correspondingly. Instead, if we increase the size of the pooling regions by 50 percent in each direction to 3×3, then the mutual exclusivity constraint now specifies that each of these edges may appear only once in a 3×3 region. As we grow a model's input image in this way, the model generates edges with less density. Of course, these issues only arise when the model must use variable amounts of pooling in order to emit a fixed-size output vector. Models that use probabilistic max pooling may still accept variably sized input images as long as the output of the model is a feature map that can scale in size proportional to the input image.

Pixels at the boundary of the image also pose some difficulty, which is exacerbated by the fact that connections in a Boltzmann machine are symmetric. If we do not implicitly zero pad the input, there will be fewer hidden units than visible units, and the visible units at the boundary of the image will not be modeled well because they lie in the receptive field of fewer hidden units. However, if we do implicitly zero pad the input, then the hidden units at the boundary will be driven by fewer input pixels and may fail to activate when needed.

20.7 Boltzmann Machines for Structured or Sequential Outputs

In the structured output scenario, we wish to train a model that can map from some input $\boldsymbol{x}$ to some output $\boldsymbol{y}$, and the different entries of $\boldsymbol{y}$ are related to each other and must obey some constraints. For example, in the speech synthesis task,

$\boldsymbol{y}$ is a waveform, and the entire waveform must sound like a coherent utterance.

A natural way to represent the relationships between the entries in $\boldsymbol{y}$ is to use a probability distribution $p(\mathbf{y} \mid \boldsymbol{x})$. Boltzmann machines, extended to model conditional distributions, can supply this probabilistic model.

The same tool of conditional modeling with a Boltzmann machine can be used not just for structured output tasks, but also for sequence modeling. In the latter case, rather than mapping an input $\boldsymbol{x}$ to an output $\boldsymbol{y}$, the model must estimate a probability distribution over a sequence of variables, $p(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(\tau)})$. Conditional Boltzmann machines can represent factors of the form $p(\mathbf{x}^{(t)} \mid \mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(t-1)})$ in order to accomplish this task.

An important sequence modeling task for the video game and film industry is modeling sequences of joint angles of skeletons used to render 3-D characters. These sequences are often collected using motion capture systems to record the movements of actors. A probabilistic model of a character's movement allows the generation of new, previously unseen, but realistic animations. To solve this sequence modeling task, Taylor *et al.* (2007) introduced a conditional RBM modeling $p(\boldsymbol{x}^{(t)} \mid \boldsymbol{x}^{(t-1)}, \ldots, \boldsymbol{x}^{(t-m)})$ for small m. The model is an RBM over $p(\boldsymbol{x}^{(t)})$ whose bias parameters are a linear function of the preceding m values of $\boldsymbol{x}$. When we condition on different values of $\boldsymbol{x}^{(t-1)}$ and earlier variables, we get a new RBM over $\mathbf{x}$. The weights in the RBM over $\mathbf{x}$ never change, but by conditioning on different past values, we can change the probability of different hidden units in the RBM being active. By activating and deactivating different subsets of hidden units, we can make large changes to the probability distribution induced on $\mathbf{x}$. Other variants of conditional RBM (Mnih *et al.*, 2011) and other variants of sequence modeling using conditional RBMs are possible (Taylor and Hinton, 2009; Sutskever *et al.*, 2009; Boulanger-Lewandowski *et al.*, 2012).

Another sequence modeling task is to model the distribution over sequences of musical notes used to compose songs. Boulanger-Lewandowski *et al.* (2012) introduced the **RNN-RBM** sequence model and applied it to this task. The RNN-RBM is a generative model of a sequence of frames $\boldsymbol{x}^{(t)}$ consisting of an RNN that emits the RBM parameters for each time step. Unlike previous approaches in which only the bias parameters of the RBM varied from one time step to the next, the RNN-RBM uses the RNN to emit all the parameters of the RBM, including the weights. To train the model, we need to be able to back-propagate the gradient of the loss function through the RNN. The loss function is not applied directly to the RNN outputs. Instead, it is applied to the RBM. This means that we must approximately differentiate the loss with respect to the RBM parameters using contrastive divergence or a related algorithm. This approximate gradient may then

be back-propagated through the RNN using the usual back-propagation through time algorithm.

20.8 Other Boltzmann Machines

Many other variants of Boltzmann machines are possible.

Boltzmann machines may be extended with different training criteria. We have focused on Boltzmann machines trained to approximately maximize the generative criterion $\log p(\boldsymbol{v})$. It is also possible to train discriminative RBMs that aim to maximize $\log p(y \mid \boldsymbol{v})$ instead (Larochelle and Bengio, 2008). This approach often performs the best when using a linear combination of both the generative and the discriminative criteria. Unfortunately, RBMs do not seem to be as powerful supervised learners as MLPs, at least using existing methodology.

Most Boltzmann machines used in practice have only second-order interactions in their energy functions, meaning that their energy functions are the sum of many terms, and each individual term includes only the product between two random variables. An example of such a term is $v_i W_{i,j} h_j$. It is also possible to train higher-order Boltzmann machines (Sejnowski, 1987) whose energy function terms involve the products between many variables. Three-way interactions between a hidden unit and two different images can model spatial transformations from one frame of video to the next (Memisevic and Hinton, 2007, 2010). Multiplication by a one-hot class variable can change the relationship between visible and hidden units depending on which class is present (Nair and Hinton, 2009). One recent example of the use of higher-order interactions is a Boltzmann machine with two groups of hidden units, one group that interacts with both the visible units $\boldsymbol{v}$ and the class label y, and another group that interacts only with the $\boldsymbol{v}$ input values (Luo *et al.*, 2011). This can be interpreted as encouraging some hidden units to learn to model the input using features that are relevant to the class, but also to learn extra hidden units that explain nuisance details necessary for the samples of $\boldsymbol{v}$ to be realistic without determining the class of the example. Another use of higher-order interactions is to gate some features. Sohn *et al.* (2013) introduced a Boltzmann machine with third-order interactions and binary mask variables associated with each visible unit. When these masking variables are set to zero, they remove the influence of a visible unit on the hidden units. This allows visible units that are not relevant to the classification problem to be removed from the inference pathway that estimates the class.

More generally, the Boltzmann machine framework is a rich space of models permitting many more model structures than have been explored so far. Developing

a new form of Boltzmann machine requires some more care and creativity than developing a new neural network layer, because it is often difficult to find an energy function that maintains tractability of all the different conditional distributions needed to use the Boltzmann machine. Despite this required effort, the field remains open to innovation.

20.9 Back-Propagation through Random Operations

Traditional neural networks implement a deterministic transformation of some input variables $\boldsymbol{x}$. When developing generative models, we often wish to extend neural networks to implement stochastic transformations of $\boldsymbol{x}$. One straightforward way to do this is to augment the neural network with extra inputs $\boldsymbol{z}$ that are sampled from some simple probability distribution, such as a uniform or Gaussian distribution. The neural network can then continue to perform deterministic computation internally, but the function $f(\boldsymbol{x}, \boldsymbol{z})$ will appear stochastic to an observer who does not have access to $\boldsymbol{z}$. Provided that f is continuous and differentiable, we can then compute the gradients necessary for training using back-propagation as usual.

As an example, let us consider the operation consisting of drawing samples y from a Gaussian distribution with mean μ and variance σ^2:

$$\mathrm{y} \sim \mathcal{N}(\mu, \sigma^2). \tag{20.54}$$

Because an individual sample of y is produced not by a function, but rather by a sampling process whose output changes every time we query it, it may seem counterintuitive to take the derivatives of y with respect to the parameters of its distribution, μ and σ^2. However, we can rewrite the sampling process as transforming an underlying random value $\mathrm{z} \sim \mathcal{N}(z; 0, 1)$ to obtain a sample from 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, as 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 section 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 and 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 = nonnegative 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 $\mathbb{E}_{\mathbf{z} \sim p(\mathbf{z})} J(f(\boldsymbol{z}; \boldsymbol{\omega}))$ 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:

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

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

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

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

Equation 20.60 relies on the assumption that J does not reference ω directly. It is trivial to extend the approach to relax this assumption. Equation 20.61 exploits the derivative rule for the logarithm, $\frac{\partial \log p(\boldsymbol{y})}{\partial \boldsymbol{\omega}} = \frac{1}{p(\boldsymbol{y})} \frac{\partial p(\boldsymbol{y})}{\partial \boldsymbol{\omega}}$. Equation 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 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 gets 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 $E_{p(\boldsymbol{y})}[J(\boldsymbol{y})\frac{\partial \log p(\boldsymbol{y})}{\partial \omega_i}^2]$ 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 equation 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 Networks

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 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 section 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 because of 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, as described in section 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 in section 20.10.7.

20.10.2 Differentiable Generator Networks

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 equation 3.47 that, for invertible, differentiable, continuous g,

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

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

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

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 section 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 doing so would mean the model could no longer be trained 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 $p_{\text{model}}(\boldsymbol{x}; g(\boldsymbol{z})) = p_{\text{model}}(\boldsymbol{x} \mid \boldsymbol{z})$. During training, however, 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 can 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 equation 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 as 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 equation 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 (section 19.4). These approaches are slow and often require the ability to compute $\mathbb{E}_{\text{z}\sim q} \log p_{\text{model}}(\boldsymbol{z}, \boldsymbol{x})$ 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. As long as $\boldsymbol{z}$ is a continuous variable, we

can then back-propagate through samples of $\boldsymbol{z}$ drawn from $q(\boldsymbol{z} \mid \boldsymbol{x}) = q(\boldsymbol{z}; f(\boldsymbol{x}; \boldsymbol{\theta}))$ 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 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 $D_{\mathrm{KL}}(p_{\mathrm{data}} \| p_{\mathrm{model}})$. As illustrated in figure 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 $p_{\mathrm{model}}(\boldsymbol{x}; g(\boldsymbol{z}))$. 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, $D_{\mathrm{KL}}(p_{\mathrm{data}} \| p_{\mathrm{model}})$, 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 were 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 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** (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 nondeterministic 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 also 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 of increasing 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 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 figure 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.

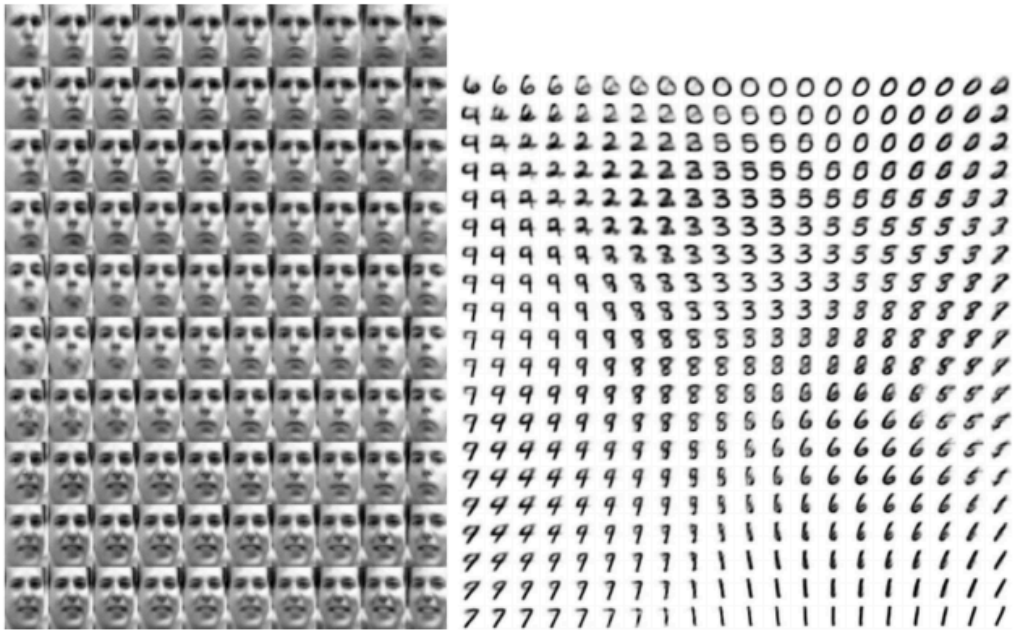

Figure 20.6: Examples of 2-D coordinate systems for high-dimensional manifolds, learned by a variational autoencoder (Kingma and Welling, 2014a). Two dimensions may be plotted directly on the page for visualization, so we can gain an understanding of how the model works by training a model with a 2-D 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 2-D 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 2-D 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. When $\max_d v(g, d)$ is convex in $\boldsymbol{\theta}^{(g)}$ (such as the case where optimization is performed directly in the space of probability density functions), 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 nonconvergence 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 its cost. It is not known to what extent this nonconvergence 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. Unfortunately, this alternative formulation does not seem to improve convergence in practice, possibly because of suboptimality of the discriminator or high variance around the expected gradient.

In realistic experiments, 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 when 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 figure 15.9. See figure 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. 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 percent of the outputs of the network as being real data. See figure 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

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).

a disadvantage, and one may also guarantee that the generator network assigns nonzero probability to all points simply by making the last layer of the generator network add Gaussian noise to all 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.

Generative moment matching 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 nonnegative 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 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 which 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**, or MMD (Schölkopf and Smola, 2002; Gretton *et al.*, 2012). 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 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 section 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 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 nonmaximal 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 section 20.10.10, 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*.

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 figure 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

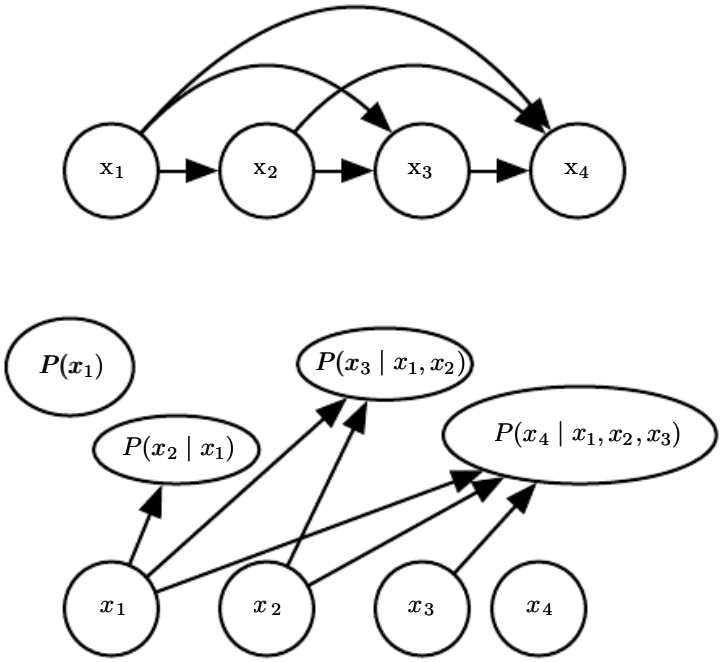

Figure 20.8: A fully visible belief network predicts the i-th variable from the $i-1$ previous ones. *(Top)*The directed graphical model for an FVBN. *(Bottom)*Corresponding computational graph for the logistic FVBN, where each prediction is made by a linear predictor.

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.

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 (figure 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 figure 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) enables 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 figure 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 multitask and transfer learning.

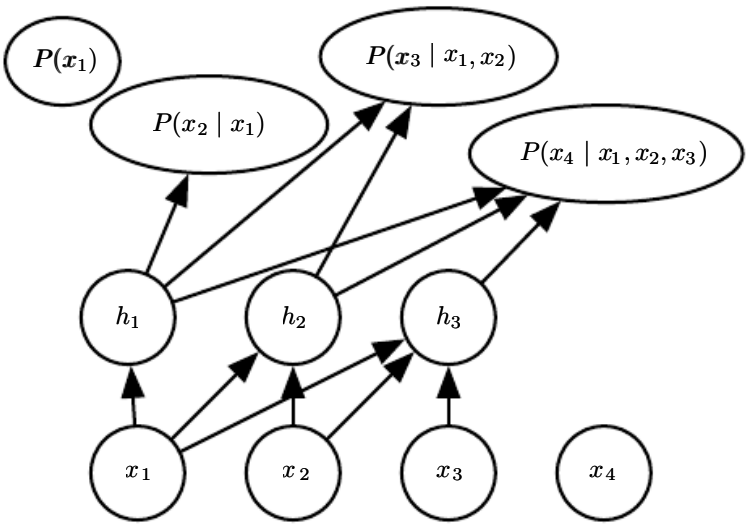

Figure 20.9: A neural auto-regressive network predicts the i-th variable x_i from the $i-1$ previous ones, but is parametrized so that features (groups of hidden units denoted h_i) that are functions of $x_1, \ldots, x_i$ can be reused in predicting all the subsequent variables $x_{i+1}, x_{i+2}, \ldots,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

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 section 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 auto-regressive 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 figure 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

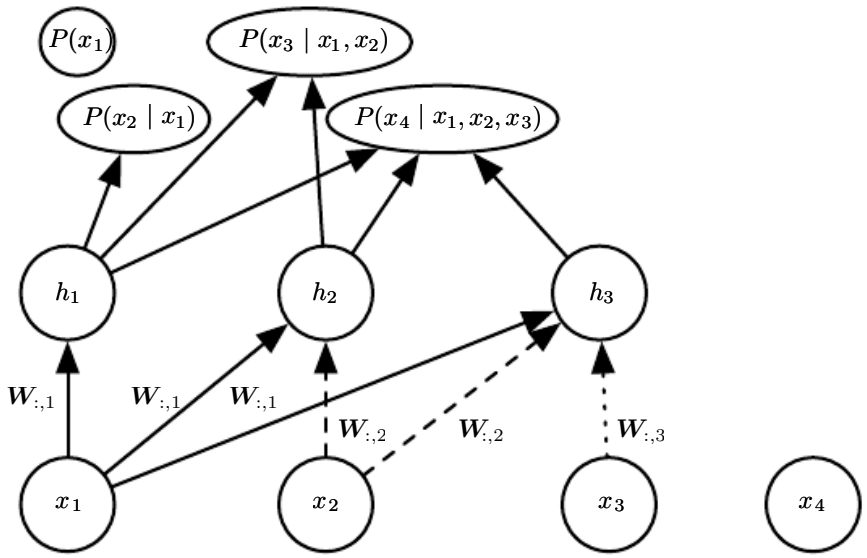

Figure 20.10: An illustration of the neural autoregressive density estimator (NADE). The hidden units are organized in groups $\boldsymbol{h}^{(j)}$ so that only the inputs $x_1, \ldots, x_i$ participate in computing $\boldsymbol{h}^{(i)}$ and predicting $P(x_j \mid x_{j-1}, \ldots, x_1)$, for $j > i$. 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 $\boldsymbol{W}_{:,i}$.

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 would loosely resemble 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 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 k steps. This approach is called NADE-k (Raiko *et al.*, 2014).

As mentioned previously, auto-regressive networks may be extended to process continuous-valued data. A particularly powerful and generic way of parametrizing a continuous density is as a Gaussian mixture (introduced in section 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 pseudogradient 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 (Uria *et al.*, 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 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 figures 20.10 and 20.9), whereas it is $O(n^2h)$ in Bengio and Bengio (2000b). For the other hidden layers, however, 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 Uria *et al.* (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 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 substeps, illustrated in figure 20.11:

1. Starting from the previous state $\boldsymbol{x}$, inject corruption noise, sampling $\tilde{\boldsymbol{x}}$ from $C(\tilde{\boldsymbol{x}} \mid \boldsymbol{x})$.

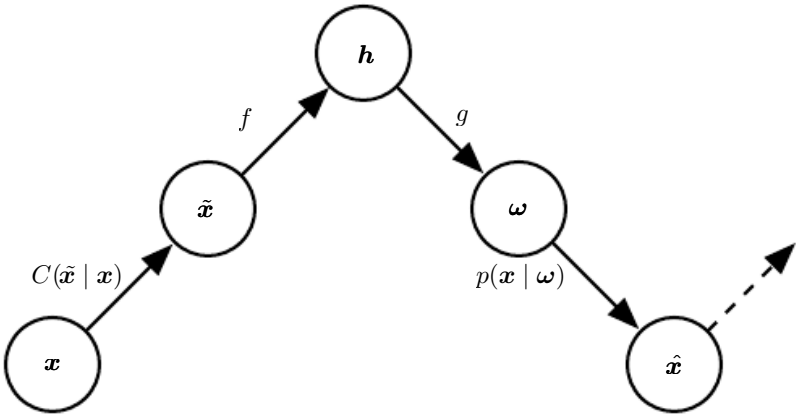

Figure 20.11: Each step of the Markov chain associated with a trained denoising autoencoder, which generates the samples from the probabilistic model implicitly trained by the denoising log-likelihood criterion. Each step consists in (a) injecting noise via corruption process C in state $\boldsymbol{x}$, 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 of adding Gaussian noise, and sampling from $p(\mathbf{x} \mid \boldsymbol{\omega})$ consists of 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).

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 $\mathbf{x}_o$ and only resampling the *free* units $\mathbf{x}_f$ given $\mathbf{x}_o$ 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 figure 20.12.

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 of alternative multiple stochastic encode-decode steps (as in the generative Markov chain), initialized at a training example (just as with the contrastive divergence algorithm, described in section 18.2), and penalizing the last probabilistic reconstructions (or all 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 further from the data can be removed more efficiently.

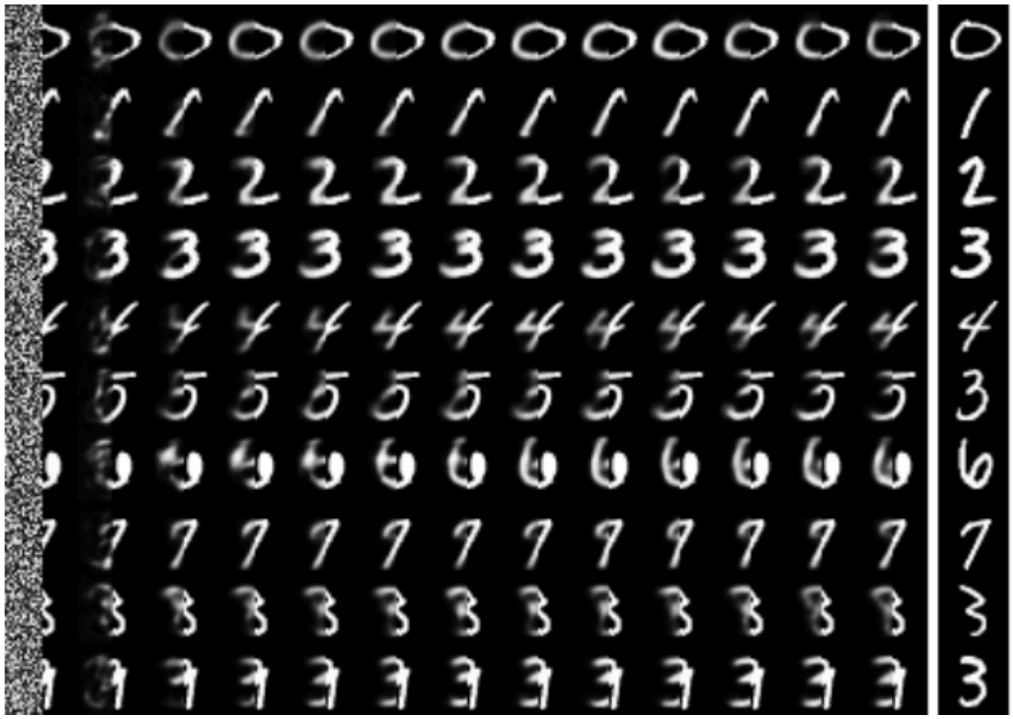

Figure 20.12: Illustration of clamping the right half of the image and running the Markov chain by resampling only the left half at each step. These samples come from a GSN trained to reconstruct MNIST digits at each time step using the walk-back procedure.

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 that 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 section 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 are 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 as 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, that is, 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 section 20.9.

The walk-back training procedure (described in section 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 enabling it 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.

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, that is, 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 nonequilibrium 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 (section 20.11.1). As with the denoising autoencoder, diffusion inversion trains a transition operator that attempts to probabilistically undo the effect of adding some noise. The difference is that diffusion inversion requires undoing only one step of the diffusion process, rather than traveling all the way back to a clean data point. 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 highly complex and multimodal). With the diffusion inversion objective, the learner can learn the shape of the density around the data points more precisely 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 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 as 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. Often, we cannot actually evaluate the log-probability of the data under the model, but can evaluate only an approximation. In these cases, it is important to think and communicate clearly about what exactly 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, for example, 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 the result of a good model or a bad AIS implementation.

Other fields of machine learning usually allow for some variation in the pre-processing 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 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 figure 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 figure 3.6. Unfortunately, even when we restrict the use of each metric to the task it is most suited for, all 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 $p_{\text{model}}(\boldsymbol{x})$ and a representation $p_{\text{model}}(\boldsymbol{h} \mid \boldsymbol{x})$, a generative model can provide answers to many inference problems about the relationships between input variables in $\boldsymbol{x}$ and can offer 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 the many different intuitive concepts they need to understand, giving them 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.