usually not very easy for a human to interpret after the fact, though visualization techniques may allow some rough characterization of what they represent. When latent variables are used in the context of traditional graphical models, they are often designed with some specific semantics in mind—the topic of a document, the intelligence of a student, the disease causing a patient's symptoms, etc. These models are often much more interpretable by human practitioners and often have more theoretical guarantees, yet are less able to scale to complex problems and are not reusable in as many different contexts as deep models.

Another obvious difference is the kind of connectivity typically used in the deep learning approach. Deep graphical models typically have large groups of units that are all connected to other groups of units, so that the interactions between two groups may be described by a single matrix. Traditional graphical models have very few connections and the choice of connections for each variable may be individually designed. The design of the model structure is tightly linked with the choice of inference algorithm. Traditional approaches to graphical models typically aim to maintain the tractability of exact inference. When this constraint is too limiting, a popular approximate inference algorithm is an algorithm called **loopy belief propagation**. Both of these approaches often work well with very sparsely connected graphs. By comparison, models used in deep learning tend to connect each visible unit v_i to very many hidden units h_j, so that $\mathbf{h}$ can provide a distributed representation of v_i (and probably several other observed variables too). Distributed representations have many advantages, but from the point of view of graphical models and computational complexity, distributed representations have the disadvantage of usually yielding graphs that are not sparse enough for the traditional techniques of exact inference and loopy belief propagation to be relevant. As a consequence, one of the most striking differences between the larger graphical models community and the deep graphical models community is that loopy belief propagation is almost never used for deep learning. Most deep models are instead designed to make Gibbs sampling or variational inference algorithms efficient. Another consideration is that deep learning models contain a very large number of latent variables, making efficient numerical code essential. This provides an additional motivation, besides the choice of high-level inference algorithm, for grouping the units into layers with a matrix describing the interaction between two layers. This allows the individual steps of the algorithm to be implemented with efficient matrix product operations, or sparsely connected generalizations, like block diagonal matrix products or convolutions.

Finally, the deep learning approach to graphical modeling is characterized by a marked tolerance of the unknown. Rather than simplifying the model until all quantities we might want can be computed exactly, we increase the power of

the model until it is just barely possible to train or use. We often use models whose marginal distributions cannot be computed, and are satisfied simply to draw approximate samples from these models. We often train models with an intractable objective function that we cannot even approximate in a reasonable amount of time, but we are still able to approximately train the model if we can efficiently obtain an estimate of the gradient of such a function. The deep learning approach is often to figure out what the minimum amount of information we absolutely need is, and then to figure out how to get a reasonable approximation of that information as quickly as possible.

16.7.1 Example: The Restricted Boltzmann Machine

The **restricted Boltzmann machine** (RBM) (Smolensky, 1986) or **harmonium** is the quintessential example of how graphical models are used for deep learning. The RBM is not itself a deep model. Instead, it has a single layer of latent variables that may be used to learn a representation for the input. In chapter 20, we will see how RBMs can be used to build many deeper models. Here, we show how the RBM exemplifies many of the practices used in a wide variety of deep graphical models: its units are organized into large groups called layers, the connectivity between layers is described by a matrix, the connectivity is relatively dense, the model is designed to allow efficient Gibbs sampling, and the emphasis of the model design is on freeing the training algorithm to learn latent variables whose semantics were not specified by the designer. Later, in section 20.2, we will revisit the RBM in more detail.

The canonical RBM is an energy-based model with binary visible and hidden units. Its energy function is

$$E(\boldsymbol{v}, \boldsymbol{h}) = -\boldsymbol{b}^\top \boldsymbol{v} - \boldsymbol{c}^\top \boldsymbol{h} - \boldsymbol{v}^\top \boldsymbol{W} \boldsymbol{h}, \tag{16.10}$$

where $\boldsymbol{b}$, $\boldsymbol{c}$, and $\boldsymbol{W}$ are unconstrained, real-valued, learnable parameters. We can see that the model is divided into two groups of units: $\boldsymbol{v}$ and $\boldsymbol{h}$, and the interaction between them is described by a matrix $\boldsymbol{W}$. The model is depicted graphically in figure 16.14. As this figure makes clear, an important aspect of this model is that there are no direct interactions between any two visible units or between any two hidden units (hence the "restricted," a general Boltzmann machine may have arbitrary connections).

The restrictions on the RBM structure yield the nice properties

$$p(\mathbf{h} \mid \mathbf{v}) = \Pi_i p(\mathrm{h}_i \mid \mathbf{v}) \tag{16.11}$$

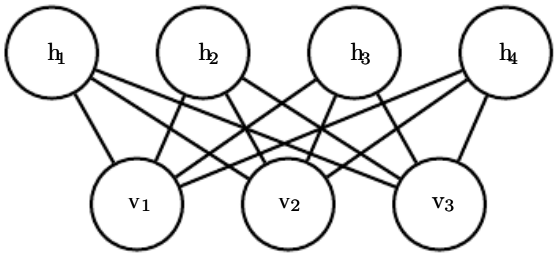

Figure 16.14: An RBM drawn as a Markov network.

and

$$p(\mathbf{v} \mid \mathbf{h}) = \Pi_i p(\mathrm{v}_i \mid \mathbf{h}). \tag{16.12}$$

The individual conditionals are simple to compute as well. For the binary RBM we obtain:

$$P(\mathrm{h}_i = 1 \mid \mathbf{v}) = \sigma\left(\mathbf{v}^\top \boldsymbol{W}_{:,i} + b_i\right), \tag{16.13}$$

$$P(\mathrm{h}_i = 0 \mid \mathbf{v}) = 1 - \sigma\left(\mathbf{v}^\top \boldsymbol{W}_{:,i} + b_i\right). \tag{16.14}$$

Together these properties allow for efficient **block Gibbs** sampling, which alternates between sampling all of $\mathbf{h}$ simultaneously and sampling all of $\mathbf{v}$ simultaneously. Samples generated by Gibbs sampling from an RBM model are shown in figure 16.15.

Since the energy function itself is just a linear function of the parameters, it is easy to take its derivatives. For example,

$$\frac{\partial}{\partial W_{i,j}} E(\mathbf{v}, \mathbf{h}) = -\mathrm{v}_i \mathrm{h}_j. \tag{16.15}$$

These two properties—efficient Gibbs sampling and efficient derivatives—make training convenient. In chapter 18, we will see that undirected models may be trained by computing such derivatives applied to samples from the model.

Training the model induces a representation $\boldsymbol{h}$ of the data $\boldsymbol{v}$. We can often use $\mathbb{E}_{\mathbf{h} \sim p(\mathbf{h} \mid \boldsymbol{v})}[\boldsymbol{h}]$ as a set of features to describe $\boldsymbol{v}$.

Overall, the RBM demonstrates the typical deep learning approach to graphical models: representation learning accomplished via layers of latent variables, combined with efficient interactions between layers parametrized by matrices.

The language of graphical models provides an elegant, flexible and clear language for describing probabilistic models. In the chapters ahead, we use this language, among other perspectives, to describe a wide variety of deep probabilistic models.

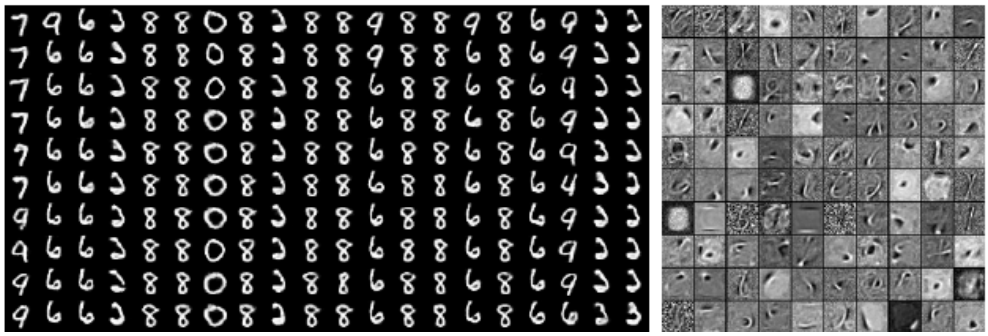

Figure 16.15: Samples from a trained RBM, and its weights. Image reproduced with permission from LISA (2008). *(Left)*Samples from a model trained on MNIST, drawn using Gibbs sampling. Each column is a separate Gibbs sampling process. Each row represents the output of another 1,000 steps of Gibbs sampling. Successive samples are highly correlated with one another. *(Right)*The corresponding weight vectors. Compare this to the samples and weights of a linear factor model, shown in figure 13.2. The samples here are much better because the RBM prior $p(\boldsymbol{h})$ is not constrained to be factorial. The RBM can learn which features should appear together when sampling. On the other hand, the RBM posterior $p(\boldsymbol{h} \mid \boldsymbol{v})$ is factorial, while the sparse coding posterior $p(\boldsymbol{h} \mid \boldsymbol{v})$ is not, so the sparse coding model may be better for feature extraction. Other models are able to have both a non-factorial $p(\boldsymbol{h})$ and a non-factorial $p(\boldsymbol{h} \mid \boldsymbol{v})$.

Chapter 17

Monte Carlo Methods

Randomized algorithms fall into two rough categories: Las Vegas algorithms and Monte Carlo algorithms. Las Vegas algorithms always return precisely the correct answer (or report that they failed). These algorithms consume a random amount of resources, usually memory or time. In contrast, Monte Carlo algorithms return answers with a random amount of error. The amount of error can typically be reduced by expending more resources (usually running time and memory). For any fixed computational budget, a Monte Carlo algorithm can provide an approximate answer.

Many problems in machine learning are so difficult that we can never expect to obtain precise answers to them. This excludes precise deterministic algorithms and Las Vegas algorithms. Instead, we must use deterministic approximate algorithms or Monte Carlo approximations. Both approaches are ubiquitous in machine learning. In this chapter, we focus on Monte Carlo methods.

17.1 Sampling and Monte Carlo Methods

Many important technologies used to accomplish machine learning goals are based on drawing samples from some probability distribution and using these samples to form a Monte Carlo estimate of some desired quantity.

17.1.1 Why Sampling?

There are many reasons that we may wish to draw samples from a probability distribution. Sampling provides a flexible way to approximate many sums and

integrals at reduced cost. Sometimes we use this to provide a significant speedup to a costly but tractable sum, as in the case when we subsample the full training cost with minibatches. In other cases, our learning algorithm requires us to approximate an intractable sum or integral, such as the gradient of the log partition function of an undirected model. In many other cases, sampling is actually our goal, in the sense that we want to train a model that can sample from the training distribution.

17.1.2 Basics of Monte Carlo Sampling

When a sum or an integral cannot be computed exactly (for example the sum has an exponential number of terms and no exact simplification is known) it is often possible to approximate it using Monte Carlo sampling. The idea is to view the sum or integral as if it was an expectation under some distribution and to *approximate the expectation by a corresponding average*. Let

$$s = \sum_{\boldsymbol{x}} p(\boldsymbol{x}) f(\boldsymbol{x}) = E_p[f(\mathbf{x})] \tag{17.1}$$

or

$$s = \int p(\boldsymbol{x}) f(\boldsymbol{x}) d\boldsymbol{x} = E_p[f(\mathbf{x})] \tag{17.2}$$

be the sum or integral to estimate, rewritten as an expectation, with the constraint that p is a probability distribution (for the sum) or a probability density (for the integral) over random variable $\mathbf{x}$.

We can approximate s by drawing n samples $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(n)}$ from p and then forming the empirical average

$$\hat{s}_n = \frac{1}{n} \sum_{i=1}^{n} f(\boldsymbol{x}^{(i)}). \tag{17.3}$$

This approximation is justified by a few different properties. The first trivial observation is that the estimator $\hat{s}$ is unbiased, since

$$\mathbb{E}[\hat{s}_n] = \frac{1}{n} \sum_{i=1}^{n} \mathbb{E}[f(\boldsymbol{x}^{(i)})] = \frac{1}{n} \sum_{i=1}^{n} s = s. \tag{17.4}$$

But in addition, the **law of large numbers** states that if the samples $\boldsymbol{x}^{(i)}$ are i.i.d., then the average converges almost surely to the expected value:

$$\lim_{n \to \infty} \hat{s}_n = s, \tag{17.5}$$

provided that the variance of the individual terms, $\text{Var}[f(\boldsymbol{x}^{(i)})]$, is bounded. To see this more clearly, consider the variance of $\hat{s}_n$ as n increases. The variance $\text{Var}[\hat{s}_n]$ decreases and converges to 0, so long as $\text{Var}[f(\mathbf{x}^{(i)})] < \infty$:

$$\text{Var}[\hat{s}_n] = \frac{1}{n^2} \sum_{i=1}^{n} \text{Var}[f(\mathbf{x})] \tag{17.6}$$

$$= \frac{\text{Var}[f(\mathbf{x})]}{n}. \tag{17.7}$$

This convenient result also tells us how to estimate the uncertainty in a Monte Carlo average or equivalently the amount of expected error of the Monte Carlo approximation. We compute both the empirical average of the $f(\boldsymbol{x}^{(i)})$ and their empirical variance,[1] and then divide the estimated variance by the number of samples n to obtain an estimator of $\text{Var}[\hat{s}_n]$. The **central limit theorem** tells us that the distribution of the average, $\hat{s}_n$, converges to a normal distribution with mean s and variance $\frac{\text{Var}[f(\mathbf{x})]}{n}$. This allows us to estimate confidence intervals around the estimate $\hat{s}_n$, using the cumulative distribution of the normal density.

However, all this relies on our ability to easily sample from the base distribution $p(\mathbf{x})$, but doing so is not always possible. When it is not feasible to sample from p, an alternative is to use importance sampling, presented in section 17.2. A more general approach is to form a sequence of estimators that converge towards the distribution of interest. That is the approach of Monte Carlo Markov chains (section 17.3).

17.2 Importance Sampling

An important step in the decomposition of the integrand (or summand) used by the Monte Carlo method in equation 17.2 is deciding which part of the integrand should play the role the probability $p(\boldsymbol{x})$ and which part of the integrand should play the role of the quantity $f(\boldsymbol{x})$ whose expected value (under that probability distribution) is to be estimated. There is no unique decomposition because $p(\boldsymbol{x})f(\boldsymbol{x})$ can always be rewritten as

$$p(\boldsymbol{x})f(\boldsymbol{x}) = q(\boldsymbol{x})\frac{p(\boldsymbol{x})f(\boldsymbol{x})}{q(\boldsymbol{x})}, \tag{17.8}$$

where we now sample from q and average $\frac{pf}{q}$. In many cases, we wish to compute an expectation for a given p and an f, and the fact that the problem is specified

[1] The unbiased estimator of the variance is often preferred, in which the sum of squared differences is divided by $n-1$ instead of n.

from the start as an expectation suggests that this p and f would be a natural choice of decomposition. However, the original specification of the problem may not be the the optimal choice in terms of the number of samples required to obtain a given level of accuracy. Fortunately, the form of the optimal choice q^* can be derived easily. The optimal q^* corresponds to what is called optimal importance sampling.

Because of the identity shown in equation 17.8, any Monte Carlo estimator

$$\hat{s}_p = \frac{1}{n} \sum_{i=1, \mathbf{x}^{(i)} \sim p}^{n} f(\boldsymbol{x}^{(i)}) \tag{17.9}$$

can be transformed into an importance sampling estimator

$$\hat{s}_q = \frac{1}{n} \sum_{i=1, \mathbf{x}^{(i)} \sim q}^{n} \frac{p(\boldsymbol{x}^{(i)}) f(\boldsymbol{x}^{(i)})}{q(\boldsymbol{x}^{(i)})}. \tag{17.10}$$

We see readily that the expected value of the estimator does not depend on q:

$$\mathbb{E}_q[\hat{s}_q] = \mathbb{E}_q[\hat{s}_p] = s. \tag{17.11}$$

However, the variance of an importance sampling estimator can be greatly sensitive to the choice of q. The variance is given by

$$\mathrm{Var}[\hat{s}_q] = \mathrm{Var}[\frac{p(\mathbf{x}) f(\mathbf{x})}{q(\mathbf{x})}]/n. \tag{17.12}$$

The minimum variance occurs when q is

$$q^*(\boldsymbol{x}) = \frac{p(\boldsymbol{x})|f(\boldsymbol{x})|}{Z}, \tag{17.13}$$

where Z is the normalization constant, chosen so that $q^*(\boldsymbol{x})$ sums or integrates to 1 as appropriate. Better importance sampling distributions put more weight where the integrand is larger. In fact, when $f(\boldsymbol{x})$ does not change sign, $\mathrm{Var}\left[\hat{s}_{q^*}\right] = 0$, meaning that *a single sample is sufficient* when the optimal distribution is used. Of course, this is only because the computation of q^* has essentially solved the original problem, so it is usually not practical to use this approach of drawing a single sample from the optimal distribution.

Any choice of sampling distribution q is valid (in the sense of yielding the correct expected value) and q^* is the optimal one (in the sense of yielding minimum variance). Sampling from q^* is usually infeasible, but other choices of q can be feasible while still reducing the variance somewhat.

Another approach is to use **biased importance sampling**, which has the advantage of not requiring normalized p or q. In the case of discrete variables, the biased importance sampling estimator is given by

$$\hat{s}_{BIS} = \frac{\sum_{i=1}^{n} \frac{p(\boldsymbol{x}^{(i)})}{q(\boldsymbol{x}^{(i)})} f(\boldsymbol{x}^{(i)})}{\sum_{i=1}^{n} \frac{p(\boldsymbol{x}^{(i)})}{q(\boldsymbol{x}^{(i)})}} \tag{17.14}$$

$$= \frac{\sum_{i=1}^{n} \frac{p(\boldsymbol{x}^{(i)})}{\tilde{q}(\boldsymbol{x}^{(i)})} f(\boldsymbol{x}^{(i)})}{\sum_{i=1}^{n} \frac{p(\boldsymbol{x}^{(i)})}{\tilde{q}(\boldsymbol{x}^{(i)})}} \tag{17.15}$$

$$= \frac{\sum_{i=1}^{n} \frac{\tilde{p}(\boldsymbol{x}^{(i)})}{\tilde{q}(\boldsymbol{x}^{(i)})} f(\boldsymbol{x}^{(i)})}{\sum_{i=1}^{n} \frac{\tilde{p}(\boldsymbol{x}^{(i)})}{\tilde{q}(\boldsymbol{x}^{(i)})}}, \tag{17.16}$$

where $\tilde{p}$ and $\tilde{q}$ are the unnormalized forms of p and q and the $\boldsymbol{x}^{(i)}$ are the samples from q. This estimator is biased because $\mathbb{E}[\hat{s}_{BIS}] \neq s$, except asymptotically when $n \to \infty$ and the denominator of equation 17.14 converges to 1. Hence this estimator is called asymptotically unbiased.

Although a good choice of q can greatly improve the efficiency of Monte Carlo estimation, a poor choice of q can make the efficiency much worse. Going back to equation 17.12, we see that if there are samples of q for which $\frac{p(\boldsymbol{x})|f(\boldsymbol{x})|}{q(\boldsymbol{x})}$ is large, then the variance of the estimator can get very large. This may happen when $q(\boldsymbol{x})$ is tiny while neither $p(\boldsymbol{x})$ nor $f(\boldsymbol{x})$ are small enough to cancel it. The q distribution is usually chosen to be a very simple distribution so that it is easy to sample from. When $\boldsymbol{x}$ is high-dimensional, this simplicity in q causes it to match p or $p|f|$ poorly. When $q(\boldsymbol{x}^{(i)}) \gg p(\boldsymbol{x}^{(i)})|f(\boldsymbol{x}^{(i)})|$, importance sampling collects useless samples (summing tiny numbers or zeros). On the other hand, when $q(\boldsymbol{x}^{(i)}) \ll p(\boldsymbol{x}^{(i)})|f(\boldsymbol{x}^{(i)})|$, which will happen more rarely, the ratio can be huge. Because these latter events are rare, they may not show up in a typical sample, yielding typical underestimation of s, compensated rarely by gross overestimation. Such very large or very small numbers are typical when $\boldsymbol{x}$ is high dimensional, because in high dimension the dynamic range of joint probabilities can be very large.

In spite of this danger, importance sampling and its variants have been found very useful in many machine learning algorithms, including deep learning algorithms. For example, see the use of importance sampling to accelerate training in neural language models with a large vocabulary (section 12.4.3.3) or other neural nets with a large number of outputs. See also how importance sampling has been used to estimate a partition function (the normalization constant of a probability

distribution) in section 18.7, and to estimate the log-likelihood in deep directed models such as the variational autoencoder, in section 20.10.3. Importance sampling may also be used to improve the estimate of the gradient of the cost function used to train model parameters with stochastic gradient descent, particularly for models such as classifiers where most of the total value of the cost function comes from a small number of misclassified examples. Sampling more difficult examples more frequently can reduce the variance of the gradient in such cases (Hinton, 2006).

17.3 Markov Chain Monte Carlo Methods

In many cases, we wish to use a Monte Carlo technique but there is no tractable method for drawing exact samples from the distribution $p_{\text{model}}(\mathbf{x})$ or from a good (low variance) importance sampling distribution $q(\mathbf{x})$. In the context of deep learning, this most often happens when $p_{\text{model}}(\mathbf{x})$ is represented by an undirected model. In these cases, we introduce a mathematical tool called a **Markov chain** to approximately sample from $p_{\text{model}}(\mathbf{x})$. The family of algorithms that use Markov chains to perform Monte Carlo estimates is called **Markov chain Monte Carlo methods** (MCMC). Markov chain Monte Carlo methods for machine learning are described at greater length in Koller and Friedman (2009). The most standard, generic guarantees for MCMC techniques are only applicable when the model does not assign zero probability to any state. Therefore, it is most convenient to present these techniques as sampling from an energy-based model (EBM) $p(\boldsymbol{x}) \propto \exp\left(-E(\boldsymbol{x})\right)$ as described in section 16.2.4. In the EBM formulation, every state is guaranteed to have non-zero probability. MCMC methods are in fact more broadly applicable and can be used with many probability distributions that contain zero probability states. However, the theoretical guarantees concerning the behavior of MCMC methods must be proven on a case-by-case basis for different families of such distributions. In the context of deep learning, it is most common to rely on the most general theoretical guarantees that naturally apply to all energy-based models.

To understand why drawing samples from an energy-based model is difficult, consider an EBM over just two variables, defining a distribution $p(\mathrm{a}, \mathrm{b})$. In order to sample a, we must draw a from $p(\mathrm{a} \mid \mathrm{b})$, and in order to sample b, we must draw it from $p(\mathrm{b} \mid \mathrm{a})$. It seems to be an intractable chicken-and-egg problem. Directed models avoid this because their graph is directed and acyclic. To perform **ancestral sampling** one simply samples each of the variables in topological order, conditioning on each variable's parents, which are guaranteed to have already been sampled (section 16.3). Ancestral sampling defines an efficient, single-pass method

of obtaining a sample.

In an EBM, we can avoid this chicken and egg problem by sampling using a Markov chain. The core idea of a Markov chain is to have a state $\boldsymbol{x}$ that begins as an arbitrary value. Over time, we randomly update $\boldsymbol{x}$ repeatedly. Eventually $\boldsymbol{x}$ becomes (very nearly) a fair sample from $p(\boldsymbol{x})$. Formally, a Markov chain is defined by a random state $\boldsymbol{x}$ and a transition distribution $T(\boldsymbol{x}' \mid \boldsymbol{x})$ specifying the probability that a random update will go to state $\boldsymbol{x}'$ if it starts in state $\boldsymbol{x}$. Running the Markov chain means repeatedly updating the state $\boldsymbol{x}$ to a value $\boldsymbol{x}'$ sampled from $T(\mathbf{x}' \mid \boldsymbol{x})$.

To gain some theoretical understanding of how MCMC methods work, it is useful to reparametrize the problem. First, we restrict our attention to the case where the random variable $\mathbf{x}$ has countably many states. We can then represent the state as just a positive integer x. Different integer values of x map back to different states $\boldsymbol{x}$ in the original problem.

Consider what happens when we run infinitely many Markov chains in parallel. All of the states of the different Markov chains are drawn from some distribution $q^{(t)}(x)$, where t indicates the number of time steps that have elapsed. At the beginning, $q^{(0)}$ is some distribution that we used to arbitrarily initialize x for each Markov chain. Later, $q^{(t)}$ is influenced by all of the Markov chain steps that have run so far. Our goal is for $q^{(t)}(x)$ to converge to $p(x)$.

Because we have reparametrized the problem in terms of positive integer x, we can describe the probability distribution q using a vector $\boldsymbol{v}$, with

$$q(\mathrm{x} = i) = v_i. \tag{17.17}$$

Consider what happens when we update a single Markov chain's state x to a new state x'. The probability of a single state landing in state x' is given by

$$q^{(t+1)}(x') = \sum_x q^{(t)}(x) T(x' \mid x). \tag{17.18}$$

Using our integer parametrization, we can represent the effect of the transition operator T using a matrix $\boldsymbol{A}$. We define $\boldsymbol{A}$ so that

$$A_{i,j} = T(\mathbf{x}' = i \mid \mathbf{x} = j). \tag{17.19}$$

Using this definition, we can now rewrite equation 17.18. Rather than writing it in terms of q and T to understand how a single state is updated, we may now use $\boldsymbol{v}$ and $\boldsymbol{A}$ to describe how the entire distribution over all the different Markov chains (running in parallel) shifts as we apply an update:

$$\boldsymbol{v}^{(t)} = \boldsymbol{A}\boldsymbol{v}^{(t-1)}. \tag{17.20}$$

Applying the Markov chain update repeatedly corresponds to multiplying by the matrix $\boldsymbol{A}$ repeatedly. In other words, we can think of the process as exponentiating the matrix $\boldsymbol{A}$:

$$\boldsymbol{v}^{(t)} = \boldsymbol{A}^t \boldsymbol{v}^{(0)}. \tag{17.21}$$

The matrix $\boldsymbol{A}$ has special structure because each of its columns represents a probability distribution. Such matrices are called **stochastic matrices**. If there is a non-zero probability of transitioning from any state x to any other state x' for some power t, then the Perron-Frobenius theorem (Perron, 1907; Frobenius, 1908) guarantees that the largest eigenvalue is real and equal to 1. Over time, we can see that all of the eigenvalues are exponentiated:

$$\boldsymbol{v}^{(t)} = \left(\boldsymbol{V}\text{diag}(\boldsymbol{\lambda})\boldsymbol{V}^{-1}\right)^t \boldsymbol{v}^{(0)} = \boldsymbol{V}\text{diag}(\boldsymbol{\lambda})^t \boldsymbol{V}^{-1}\boldsymbol{v}^{(0)}. \tag{17.22}$$

This process causes all of the eigenvalues that are not equal to 1 to decay to zero. Under some additional mild conditions, $\boldsymbol{A}$ is guaranteed to have only one eigenvector with eigenvalue 1. The process thus converges to a **stationary distribution**, sometimes also called the **equilibrium distribution**. At convergence,

$$\boldsymbol{v}' = \boldsymbol{A}\boldsymbol{v} = \boldsymbol{v}, \tag{17.23}$$

and this same condition holds for every additional step. This is an eigenvector equation. To be a stationary point, $\boldsymbol{v}$ must be an eigenvector with corresponding eigenvalue 1. This condition guarantees that once we have reached the stationary distribution, repeated applications of the transition sampling procedure do not change the *distribution* over the states of all the various Markov chains (although transition operator does change each individual state, of course).

If we have chosen T correctly, then the stationary distribution q will be equal to the distribution p we wish to sample from. We will describe how to choose T shortly, in section 17.4.

Most properties of Markov Chains with countable states can be generalized to continuous variables. In this situation, some authors call the Markov Chain a **Harris chain** but we use the term Markov Chain to describe both conditions. In general, a Markov chain with transition operator T will converge, under mild conditions, to a fixed point described by the equation

$$q'(\mathbf{x}') = \mathbb{E}_{\mathbf{x}\sim q} T(\mathbf{x}' \mid \mathbf{x}), \tag{17.24}$$

which in the discrete case is just rewriting equation 17.23. When $\mathbf{x}$ is discrete, the expectation corresponds to a sum, and when $\mathbf{x}$ is continuous, the expectation corresponds to an integral.

Regardless of whether the state is continuous or discrete, all Markov chain methods consist of repeatedly applying stochastic updates until eventually the state begins to yield samples from the equilibrium distribution. Running the Markov chain until it reaches its equilibrium distribution is called "**burning in**" the Markov chain. After the chain has reached equilibrium, a sequence of infinitely many samples may be drawn from from the equilibrium distribution. They are identically distributed but any two successive samples will be highly correlated with each other. A finite sequence of samples may thus not be very representative of the equilibrium distribution. One way to mitigate this problem is to return only every n successive samples, so that our estimate of the statistics of the equilibrium distribution is not as biased by the correlation between an MCMC sample and the next several samples. Markov chains are thus expensive to use because of the time required to burn in to the equilibrium distribution and the time required to transition from one sample to another reasonably decorrelated sample after reaching equilibrium. If one desires truly independent samples, one can run multiple Markov chains in parallel. This approach uses extra parallel computation to eliminate latency. The strategy of using only a single Markov chain to generate all samples and the strategy of using one Markov chain for each desired sample are two extremes; deep learning practitioners usually use a number of chains that is similar to the number of examples in a minibatch and then draw as many samples as are needed from this fixed set of Markov chains. A commonly used number of Markov chains is 100.

Another difficulty is that we do not know in advance how many steps the Markov chain must run before reaching its equilibrium distribution. This length of time is called the **mixing time**. It is also very difficult to test whether a Markov chain has reached equilibrium. We do not have a precise enough theory for guiding us in answering this question. Theory tells us that the chain will converge, but not much more. If we analyze the Markov chain from the point of view of a matrix $\boldsymbol{A}$ acting on a vector of probabilities $\boldsymbol{v}$, then we know that the chain mixes when $\boldsymbol{A}^t$ has effectively lost all of the eigenvalues from $\boldsymbol{A}$ besides the unique eigenvalue of 1. This means that the magnitude of the second largest eigenvalue will determine the mixing time. However, in practice, we cannot actually represent our Markov chain in terms of a matrix. The number of states that our probabilistic model can visit is exponentially large in the number of variables, so it is infeasible to represent $\boldsymbol{v}$, $\boldsymbol{A}$, or the eigenvalues of $\boldsymbol{A}$. Due to these and other obstacles, we usually do not know whether a Markov chain has mixed. Instead, we simply run the Markov chain for an amount of time that we roughly estimate to be sufficient, and use heuristic methods to determine whether the chain has mixed. These heuristic methods include manually inspecting samples or measuring correlations between

successive samples.

17.4 Gibbs Sampling

So far we have described how to draw samples from a distribution $q(\boldsymbol{x})$ by repeatedly updating $\boldsymbol{x} \leftarrow \boldsymbol{x}' \sim T(\boldsymbol{x}' \mid \boldsymbol{x})$. However, we have not described how to ensure that $q(\boldsymbol{x})$ is a useful distribution. Two basic approaches are considered in this book. The first one is to derive T from a given learned p_{model}, described below with the case of sampling from EBMs. The second one is to directly parametrize T and learn it, so that its stationary distribution implicitly defines the p_{model} of interest. Examples of this second approach are discussed in sections 20.12 and 20.13.

In the context of deep learning, we commonly use Markov chains to draw samples from an energy-based model defining a distribution $p_{\text{model}}(\boldsymbol{x})$. In this case, we want the $q(\boldsymbol{x})$ for the Markov chain to be $p_{\text{model}}(\boldsymbol{x})$. To obtain the desired $q(\boldsymbol{x})$, we must choose an appropriate $T(\boldsymbol{x}' \mid \boldsymbol{x})$.

A conceptually simple and effective approach to building a Markov chain that samples from $p_{\text{model}}(\boldsymbol{x})$ is to use **Gibbs sampling**, in which sampling from $T(\mathbf{x}' \mid \mathbf{x})$ is accomplished by selecting one variable x_i and sampling it from p_{model} conditioned on its neighbors in the undirected graph $\mathcal{G}$ defining the structure of the energy-based model. It is also possible to sample several variables at the same time so long as they are conditionally independent given all of their neighbors. As shown in the RBM example in section 16.7.1, all of the hidden units of an RBM may be sampled simultaneously because they are conditionally independent from each other given all of the visible units. Likewise, all of the visible units may be sampled simultaneously because they are conditionally independent from each other given all of the hidden units. Gibbs sampling approaches that update many variables simultaneously in this way are called **block Gibbs sampling**.

Alternate approaches to designing Markov chains to sample from p_{model} are possible. For example, the Metropolis-Hastings algorithm is widely used in other disciplines. In the context of the deep learning approach to undirected modeling, it is rare to use any approach other than Gibbs sampling. Improved sampling techniques are one possible research frontier.

17.5 The Challenge of Mixing between Separated Modes

The primary difficulty involved with MCMC methods is that they have a tendency to **mix** poorly. Ideally, successive samples from a Markov chain designed to sample

from $p(\boldsymbol{x})$ would be completely independent from each other and would visit many different regions in $\boldsymbol{x}$ space proportional to their probability. Instead, especially in high dimensional cases, MCMC samples become very correlated. We refer to such behavior as slow mixing or even failure to mix. MCMC methods with slow mixing can be seen as inadvertently performing something resembling noisy gradient descent on the energy function, or equivalently noisy hill climbing on the probability, with respect to the state of the chain (the random variables being sampled). The chain tends to take small steps (in the space of the state of the Markov chain), from a configuration $\boldsymbol{x}^{(t-1)}$ to a configuration $\boldsymbol{x}^{(t)}$, with the energy $E(\boldsymbol{x}^{(t)})$ generally lower or approximately equal to the energy $E(\boldsymbol{x}^{(t-1)})$, with a preference for moves that yield lower energy configurations. When starting from a rather improbable configuration (higher energy than the typical ones from $p(\mathbf{x})$), the chain tends to gradually reduce the energy of the state and only occasionally move to another mode. Once the chain has found a region of low energy (for example, if the variables are pixels in an image, a region of low energy might be a connected manifold of images of the same object), which we call a mode, the chain will tend to walk around that mode (following a kind of random walk). Once in a while it will step out of that mode and generally return to it or (if it finds an escape route) move towards another mode. The problem is that successful escape routes are rare for many interesting distributions, so the Markov chain will continue to sample the same mode longer than it should.

This is very clear when we consider the Gibbs sampling algorithm (section 17.4). In this context, consider the probability of going from one mode to a nearby mode within a given number of steps. What will determine that probability is the shape of the "energy barrier" between these modes. Transitions between two modes that are separated by a high energy barrier (a region of low probability) are exponentially less likely (in terms of the height of the energy barrier). This is illustrated in figure 17.1. The problem arises when there are multiple modes with high probability that are separated by regions of low probability, especially when each Gibbs sampling step must update only a small subset of variables whose values are largely determined by the other variables.

As a simple example, consider an energy-based model over two variables a and b, which are both binary with a sign, taking on values -1 and 1. If $E(\mathrm{a}, \mathrm{b}) = -w\mathrm{ab}$ for some large positive number w, then the model expresses a strong belief that a and b have the same sign. Consider updating b using a Gibbs sampling step with $\mathrm{a} = 1$. The conditional distribution over b is given by $P(\mathrm{b} = 1 \mid \mathrm{a} = 1) = \sigma(w)$. If w is large, the sigmoid saturates, and the probability of also assigning b to be 1 is close to 1. Likewise, if $\mathrm{a} = -1$, the probability of assigning b to be -1 is close to 1. According to $P_{\text{model}}(\mathrm{a}, \mathrm{b})$, both signs of both variables are equally likely.

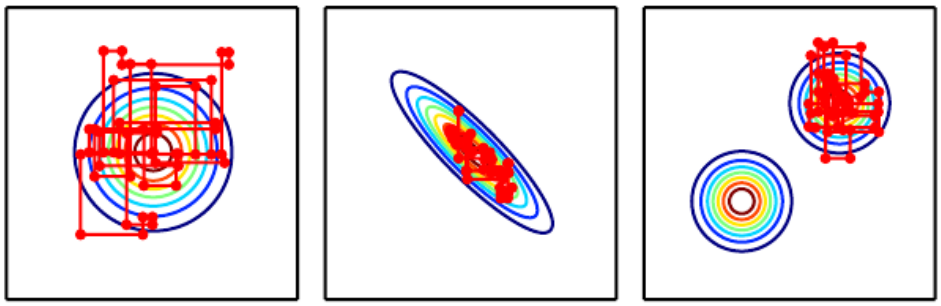

Figure 17.1: Paths followed by Gibbs sampling for three distributions, with the Markov chain initialized at the mode in both cases. *(Left)*A multivariate normal distribution with two independent variables. Gibbs sampling mixes well because the variables are independent. *(Center)*A multivariate normal distribution with highly correlated variables. The correlation between variables makes it difficult for the Markov chain to mix. Because the update for each variable must be conditioned on the other variable, the correlation reduces the rate at which the Markov chain can move away from the starting point. *(Right)*A mixture of Gaussians with widely separated modes that are not axis-aligned. Gibbs sampling mixes very slowly because it is difficult to change modes while altering only one variable at a time.

According to $P_{\text{model}}(\text{a} \mid \text{b})$, both variables should have the same sign. This means that Gibbs sampling will only very rarely flip the signs of these variables.

In more practical scenarios, the challenge is even greater because we care not only about making transitions between two modes but more generally between all the many modes that a real model might contain. If several such transitions are difficult because of the difficulty of mixing between modes, then it becomes very expensive to obtain a reliable set of samples covering most of the modes, and convergence of the chain to its stationary distribution is very slow.

Sometimes this problem can be resolved by finding groups of highly dependent units and updating all of them simultaneously in a block. Unfortunately, when the dependencies are complicated, it can be computationally intractable to draw a sample from the group. After all, the problem that the Markov chain was originally introduced to solve is this problem of sampling from a large group of variables.

In the context of models with latent variables, which define a joint distribution $p_{\text{model}}(\boldsymbol{x}, \boldsymbol{h})$, we often draw samples of $\boldsymbol{x}$ by alternating between sampling from $p_{\text{model}}(\boldsymbol{x} \mid \boldsymbol{h})$ and sampling from $p_{\text{model}}(\boldsymbol{h} \mid \boldsymbol{x})$. From the point of view of mixing

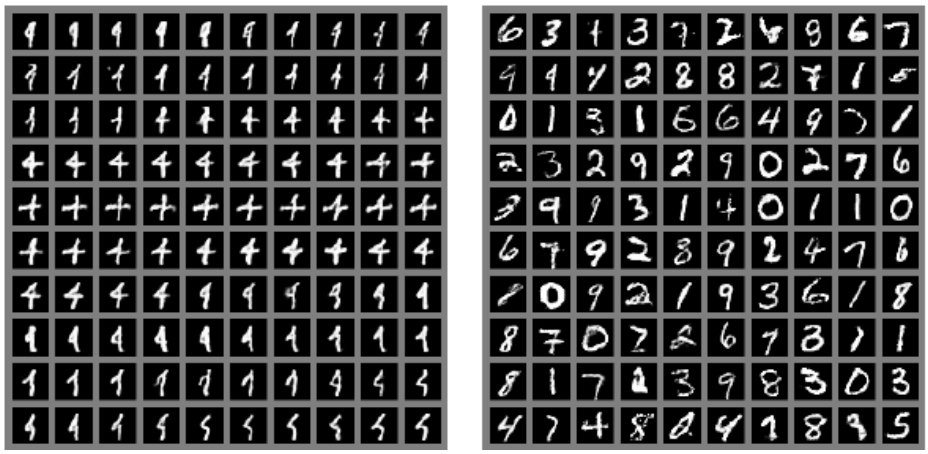

Figure 17.2: An illustration of the slow mixing problem in deep probabilistic models. Each panel should be read left to right, top to bottom. *(Left)*Consecutive samples from Gibbs sampling applied to a deep Boltzmann machine trained on the MNIST dataset. Consecutive samples are similar to each other. Because the Gibbs sampling is performed in a deep graphical model, this similarity is based more on semantic rather than raw visual features, but it is still difficult for the Gibbs chain to transition from one mode of the distribution to another, for example by changing the digit identity. *(Right)*Consecutive ancestral samples from a generative adversarial network. Because ancestral sampling generates each sample independently from the others, there is no mixing problem.

rapidly, we would like $p_{\text{model}}(\boldsymbol{h} \mid \boldsymbol{x})$ to have very high entropy. However, from the point of view of learning a useful representation of $\boldsymbol{h}$, we would like $\boldsymbol{h}$ to encode enough information about $\boldsymbol{x}$ to reconstruct it well, which implies that $\boldsymbol{h}$ and $\boldsymbol{x}$ should have very high mutual information. These two goals are at odds with each other. We often learn generative models that very precisely encode $\boldsymbol{x}$ into $\boldsymbol{h}$ but are not able to mix very well. This situation arises frequently with Boltzmann machines—the sharper the distribution a Boltzmann machine learns, the harder it is for a Markov chain sampling from the model distribution to mix well. This problem is illustrated in figure 17.2.

All this could make MCMC methods less useful when the distribution of interest has a manifold structure with a separate manifold for each class: the distribution is concentrated around many modes and these modes are separated by vast regions of high energy. This type of distribution is what we expect in many classification problems and would make MCMC methods converge very slowly because of poor mixing between modes.

17.5.1 Tempering to Mix between Modes

When a distribution has sharp peaks of high probability surrounded by regions of low probability, it is difficult to mix between the different modes of the distribution. Several techniques for faster mixing are based on constructing alternative versions of the target distribution in which the peaks are not as high and the surrounding valleys are not as low. Energy-based models provide a particularly simple way to do so. So far, we have described an energy-based model as defining a probability distribution

$$p(\boldsymbol{x}) \propto \exp\left(-E(\boldsymbol{x})\right). \tag{17.25}$$

Energy-based models may be augmented with an extra parameter β controlling how sharply peaked the distribution is:

$$p_\beta(\boldsymbol{x}) \propto \exp\left(-\beta E(\boldsymbol{x})\right). \tag{17.26}$$

The β parameter is often described as being the reciprocal of the **temperature**, reflecting the origin of energy-based models in statistical physics. When the temperature falls to zero and β rises to infinity, the energy-based model becomes deterministic. When the temperature rises to infinity and β falls to zero, the distribution (for discrete $\boldsymbol{x}$) becomes uniform.

Typically, a model is trained to be evaluated at $\beta = 1$. However, we can make use of other temperatures, particularly those where $\beta < 1$. **Tempering** is a general strategy of mixing between modes of p_1 rapidly by drawing samples with $\beta < 1$.

Markov chains based on **tempered transitions** (Neal, 1994) temporarily sample from higher-temperature distributions in order to mix to different modes, then resume sampling from the unit temperature distribution. These techniques have been applied to models such as RBMs (Salakhutdinov, 2010). Another approach is to use **parallel tempering** (Iba, 2001), in which the Markov chain simulates many different states in parallel, at different temperatures. The highest temperature states mix slowly, while the lowest temperature states, at temperature 1, provide accurate samples from the model. The transition operator includes stochastically swapping states between two different temperature levels, so that a sufficiently high-probability sample from a high-temperature slot can jump into a lower temperature slot. This approach has also been applied to RBMs (Desjardins *et al.*, 2010; Cho *et al.*, 2010). Although tempering is a promising approach, at this point it has not allowed researchers to make a strong advance in solving the challenge of sampling from complex EBMs. One possible reason is that there are **critical temperatures** around which the temperature transition must be very slow (as the temperature is gradually reduced) in order for tempering to be effective.

17.5.2 Depth May Help Mixing

When drawing samples from a latent variable model $p(\boldsymbol{h}, \boldsymbol{x})$, we have seen that if $p(\boldsymbol{h} \mid \boldsymbol{x})$ encodes $\boldsymbol{x}$ too well, then sampling from $p(\boldsymbol{x} \mid \boldsymbol{h})$ will not change $\boldsymbol{x}$ very much and mixing will be poor. One way to resolve this problem is to make $\boldsymbol{h}$ be a deep representation, that encodes $\boldsymbol{x}$ into $\boldsymbol{h}$ in such a way that a Markov chain in the space of $\boldsymbol{h}$ can mix more easily. Many representation learning algorithms, such as autoencoders and RBMs, tend to yield a marginal distribution over $\boldsymbol{h}$ that is more uniform and more unimodal than the original data distribution over $\boldsymbol{x}$. It can be argued that this arises from trying to minimize reconstruction error while using all of the available representation space, because minimizing reconstruction error over the training examples will be better achieved when different training examples are easily distinguishable from each other in $\boldsymbol{h}$-space, and thus well separated. Bengio *et al.* (2013a) observed that deeper stacks of regularized autoencoders or RBMs yield marginal distributions in the top-level $\boldsymbol{h}$-space that appeared more spread out and more uniform, with less of a gap between the regions corresponding to different modes (categories, in the experiments). Training an RBM in that higher-level space allowed Gibbs sampling to mix faster between modes. It remains however unclear how to exploit this observation to help better train and sample from deep generative models.

Despite the difficulty of mixing, Monte Carlo techniques are useful and are often the best tool available. Indeed, they are the primary tool used to confront the intractable partition function of undirected models, discussed next.

Chapter 18

Confronting the Partition Function

In section 16.2.2 we saw that many probabilistic models (commonly known as undirected graphical models) are defined by an unnormalized probability distribution $\tilde{p}(\mathbf{x}; \theta)$. We must normalize $\tilde{p}$ by dividing by a partition function $Z(\boldsymbol{\theta})$ in order to obtain a valid probability distribution:

$$p(\mathbf{x}; \boldsymbol{\theta}) = \frac{1}{Z(\boldsymbol{\theta})} \tilde{p}(\mathbf{x}; \boldsymbol{\theta}). \tag{18.1}$$

The partition function is an integral (for continuous variables) or sum (for discrete variables) over the unnormalized probability of all states:

$$\int \tilde{p}(\boldsymbol{x}) d\boldsymbol{x} \tag{18.2}$$

or

$$\sum_{\boldsymbol{x}} \tilde{p}(\boldsymbol{x}). \tag{18.3}$$

This operation is intractable for many interesting models.

As we will see in chapter 20, several deep learning models are designed to have a tractable normalizing constant, or are designed to be used in ways that do not involve computing $p(\mathbf{x})$ at all. However, other models directly confront the challenge of intractable partition functions. In this chapter, we describe techniques used for training and evaluating models that have intractable partition functions.

18.1 The Log-Likelihood Gradient

What makes learning undirected models by maximum likelihood particularly difficult is that the partition function depends on the parameters. The gradient of the log-likelihood with respect to the parameters has a term corresponding to the gradient of the partition function:

$$\nabla_{\boldsymbol{\theta}} \log p(\mathbf{x}; \boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} \log \tilde{p}(\mathbf{x}; \boldsymbol{\theta}) - \nabla_{\boldsymbol{\theta}} \log Z(\boldsymbol{\theta}). \tag{18.4}$$

This is a well-known decomposition into the **positive phase** and **negative phase** of learning.

For most undirected models of interest, the negative phase is difficult. Models with no latent variables or with few interactions between latent variables typically have a tractable positive phase. The quintessential example of a model with a straightforward positive phase and difficult negative phase is the RBM, which has hidden units that are conditionally independent from each other given the visible units. The case where the positive phase is difficult, with complicated interactions between latent variables, is primarily covered in chapter 19. This chapter focuses on the difficulties of the negative phase.

Let us look more closely at the gradient of $\log Z$:

$$\nabla_{\boldsymbol{\theta}} \log Z \tag{18.5}$$

$$= \frac{\nabla_{\boldsymbol{\theta}} Z}{Z} \tag{18.6}$$

$$= \frac{\nabla_{\boldsymbol{\theta}} \sum_{\mathbf{x}} \tilde{p}(\mathbf{x})}{Z} \tag{18.7}$$

$$= \frac{\sum_{\mathbf{x}} \nabla_{\boldsymbol{\theta}} \tilde{p}(\mathbf{x})}{Z}. \tag{18.8}$$

For models that guarantee $p(\mathbf{x}) > 0$ for all $\mathbf{x}$, we can substitute $\exp(\log \tilde{p}(\mathbf{x}))$ for $\tilde{p}(\mathbf{x})$:

$$\frac{\sum_{\mathbf{x}} \nabla_{\boldsymbol{\theta}} \exp(\log \tilde{p}(\mathbf{x}))}{Z} \tag{18.9}$$

$$= \frac{\sum_{\mathbf{x}} \exp(\log \tilde{p}(\mathbf{x})) \nabla_{\boldsymbol{\theta}} \log \tilde{p}(\mathbf{x})}{Z} \tag{18.10}$$

$$= \frac{\sum_{\mathbf{x}} \tilde{p}(\mathbf{x}) \nabla_{\boldsymbol{\theta}} \log \tilde{p}(\mathbf{x})}{Z} \tag{18.11}$$

$$= \sum_{\mathbf{x}} p(\mathbf{x}) \nabla_{\boldsymbol{\theta}} \log \tilde{p}(\mathbf{x}) \tag{18.12}$$

$$= \mathbb{E}_{\mathbf{x} \sim p(\mathbf{x})} \nabla_{\boldsymbol{\theta}} \log \tilde{p}(\mathbf{x}). \tag{18.13}$$

This derivation made use of summation over discrete $\boldsymbol{x}$, but a similar result applies using integration over continuous $\boldsymbol{x}$. In the continuous version of the derivation, we use Leibniz's rule for differentiation under the integral sign to obtain the identity

$$\nabla_{\boldsymbol{\theta}} \int \tilde{p}(\mathbf{x}) d\boldsymbol{x} = \int \nabla_{\boldsymbol{\theta}} \tilde{p}(\mathbf{x}) d\boldsymbol{x}. \tag{18.14}$$

This identity is applicable only under certain regularity conditions on $\tilde{p}$ and $\nabla_{\boldsymbol{\theta}} \tilde{p}(\mathbf{x})$. In measure theoretic terms, the conditions are: (i) The unnormalized distribution $\tilde{p}$ must be a Lebesgue-integrable function of $\boldsymbol{x}$ for every value of $\boldsymbol{\theta}$; (ii) The gradient $\nabla_{\boldsymbol{\theta}} \tilde{p}(\mathbf{x})$ must exist for all $\boldsymbol{\theta}$ and almost all $\boldsymbol{x}$; (iii) There must exist an integrable function $R(\boldsymbol{x})$ that bounds $\nabla_{\boldsymbol{\theta}} \tilde{p}(\mathbf{x})$ in the sense that $\max_i |\frac{\partial}{\partial \theta_i} \tilde{p}(\mathbf{x})| \leq R(\boldsymbol{x})$ for all $\boldsymbol{\theta}$ and almost all $\boldsymbol{x}$. Fortunately, most machine learning models of interest have these properties.

This identity

$$\nabla_{\boldsymbol{\theta}} \log Z = \mathbb{E}_{\mathbf{x} \sim p(\mathbf{x})} \nabla_{\boldsymbol{\theta}} \log \tilde{p}(\mathbf{x}) \tag{18.15}$$

is the basis for a variety of Monte Carlo methods for approximately maximizing the likelihood of models with intractable partition functions.

The Monte Carlo approach to learning undirected models provides an intuitive framework in which we can think of both the positive phase and the negative phase. In the positive phase, we increase $\log \tilde{p}(\mathbf{x})$ for $\boldsymbol{x}$ drawn from the data. In the negative phase, we decrease the partition function by decreasing $\log \tilde{p}(\mathbf{x})$ drawn from the model distribution.

In the deep learning literature, it is common to parametrize $\log \tilde{p}$ in terms of an energy function (equation 16.7). In this case, we can interpret the positive phase as pushing down on the energy of training examples and the negative phase as pushing up on the energy of samples drawn from the model, as illustrated in figure 18.1.

18.2 Stochastic Maximum Likelihood and Contrastive Divergence

The naive way of implementing equation 18.15 is to compute it by burning in a set of Markov chains from a random initialization every time the gradient is needed. When learning is performed using stochastic gradient descent, this means the chains must be burned in once per gradient step. This approach leads to the

training procedure presented in algorithm 18.1. The high cost of burning in the Markov chains in the inner loop makes this procedure computationally infeasible, but this procedure is the starting point that other more practical algorithms aim to approximate.

Algorithm 18.1 A naive MCMC algorithm for maximizing the log-likelihood with an intractable partition function using gradient ascent.

Set ϵ, the step size, to a small positive number.
Set k, the number of Gibbs steps, high enough to allow burn in. Perhaps 100 to train an RBM on a small image patch.
while not converged **do**
  Sample a minibatch of m examples $\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\}$ from the training set.
  $\mathbf{g} \leftarrow \frac{1}{m} \sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} \log \tilde{p}(\mathbf{x}^{(i)}; \boldsymbol{\theta})$.
  Initialize a set of m samples $\{\tilde{\mathbf{x}}^{(1)}, \ldots, \tilde{\mathbf{x}}^{(m)}\}$ to random values (e.g., from a uniform or normal distribution, or possibly a distribution with marginals matched to the model's marginals).
  for $i = 1$ to k **do**
    for $j = 1$ to m **do**
      $\tilde{\mathbf{x}}^{(j)} \leftarrow \text{gibbs\_update}(\tilde{\mathbf{x}}^{(j)})$.
    end for
  end for
  $\mathbf{g} \leftarrow \mathbf{g} - \frac{1}{m} \sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} \log \tilde{p}(\tilde{\mathbf{x}}^{(i)}; \boldsymbol{\theta})$.
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \epsilon \mathbf{g}$.
end while

We can view the MCMC approach to maximum likelihood as trying to achieve balance between two forces, one pushing up on the model distribution where the data occurs, and another pushing down on the model distribution where the model samples occur. Figure 18.1 illustrates this process. The two forces correspond to maximizing $\log \tilde{p}$ and minimizing $\log Z$. Several approximations to the negative phase are possible. Each of these approximations can be understood as making the negative phase computationally cheaper but also making it push down in the wrong locations.

Because the negative phase involves drawing samples from the model's distribution, we can think of it as finding points that the model believes in strongly. Because the negative phase acts to reduce the probability of those points, they are generally considered to represent the model's incorrect beliefs about the world. They are frequently referred to in the literature as "hallucinations" or "fantasy particles." In fact, the negative phase has been proposed as a possible explanation

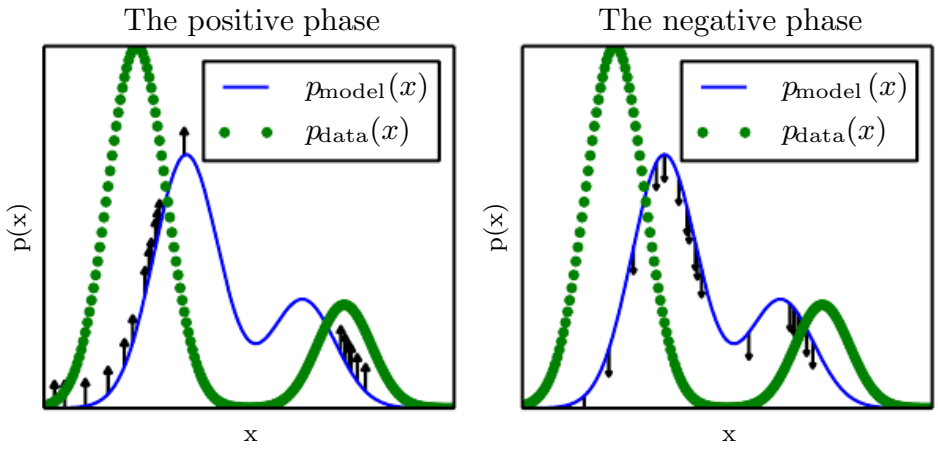

Figure 18.1: The view of algorithm 18.1 as having a "positive phase" and "negative phase." *(Left)*In the positive phase, we sample points from the data distribution, and push up on their unnormalized probability. This means points that are likely in the data get pushed up on more. *(Right)*In the negative phase, we sample points from the model distribution, and push down on their unnormalized probability. This counteracts the positive phase's tendency to just add a large constant to the unnormalized probability everywhere. When the data distribution and the model distribution are equal, the positive phase has the same chance to push up at a point as the negative phase has to push down. When this occurs, there is no longer any gradient (in expectation) and training must terminate.

for dreaming in humans and other animals (Crick and Mitchison, 1983), the idea being that the brain maintains a probabilistic model of the world and follows the gradient of $\log \tilde{p}$ while experiencing real events while awake and follows the negative gradient of $\log \tilde{p}$ to minimize $\log Z$ while sleeping and experiencing events sampled from the current model. This view explains much of the language used to describe algorithms with a positive and negative phase, but it has not been proven to be correct with neuroscientific experiments. In machine learning models, it is usually necessary to use the positive and negative phase simultaneously, rather than in separate time periods of wakefulness and REM sleep. As we will see in section 19.5, other machine learning algorithms draw samples from the model distribution for other purposes and such algorithms could also provide an account for the function of dream sleep.

Given this understanding of the role of the positive and negative phase of learning, we can attempt to design a less expensive alternative to algorithm 18.1. The main cost of the naive MCMC algorithm is the cost of burning in the Markov chains from a random initialization at each step. A natural solution is to initialize the Markov chains from a distribution that is very close to the model distribution,

so that the burn in operation does not take as many steps.

The **contrastive divergence** (CD, or CD-k to indicate CD with k Gibbs steps) algorithm initializes the Markov chain at each step with samples from the data distribution (Hinton, 2000, 2010). This approach is presented as algorithm 18.2. Obtaining samples from the data distribution is free, because they are already available in the data set. Initially, the data distribution is not close to the model distribution, so the negative phase is not very accurate. Fortunately, the positive phase can still accurately increase the model's probability of the data. After the positive phase has had some time to act, the model distribution is closer to the data distribution, and the negative phase starts to become accurate.

Algorithm 18.2 The contrastive divergence algorithm, using gradient ascent as the optimization procedure.

Set ϵ, the step size, to a small positive number.
Set k, the number of Gibbs steps, high enough to allow a Markov chain sampling from $p(\mathbf{x};\boldsymbol{\theta})$ to mix when initialized from p_{data}. Perhaps 1-20 to train an RBM on a small image patch.
while not converged **do**
  Sample a minibatch of m examples $\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\}$ from the training set.
  $\mathbf{g} \leftarrow \frac{1}{m} \sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} \log \tilde{p}(\mathbf{x}^{(i)};\boldsymbol{\theta})$.
  for $i = 1$ to m **do**
    $\tilde{\mathbf{x}}^{(i)} \leftarrow \mathbf{x}^{(i)}$.
  end for
  for $i = 1$ to k **do**
    for $j = 1$ to m **do**
      $\tilde{\mathbf{x}}^{(j)} \leftarrow \text{gibbs\_update}(\tilde{\mathbf{x}}^{(j)})$.
    end for
  end for
  $\mathbf{g} \leftarrow \mathbf{g} - \frac{1}{m} \sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} \log \tilde{p}(\tilde{\mathbf{x}}^{(i)};\boldsymbol{\theta})$.
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \epsilon \mathbf{g}$.
end while

Of course, CD is still an approximation to the correct negative phase. The main way that CD qualitatively fails to implement the correct negative phase is that it fails to suppress regions of high probability that are far from actual training examples. These regions that have high probability under the model but low probability under the data generating distribution are called **spurious modes**. Figure 18.2 illustrates why this happens. Essentially, it is because modes in the model distribution that are far from the data distribution will not be visited by

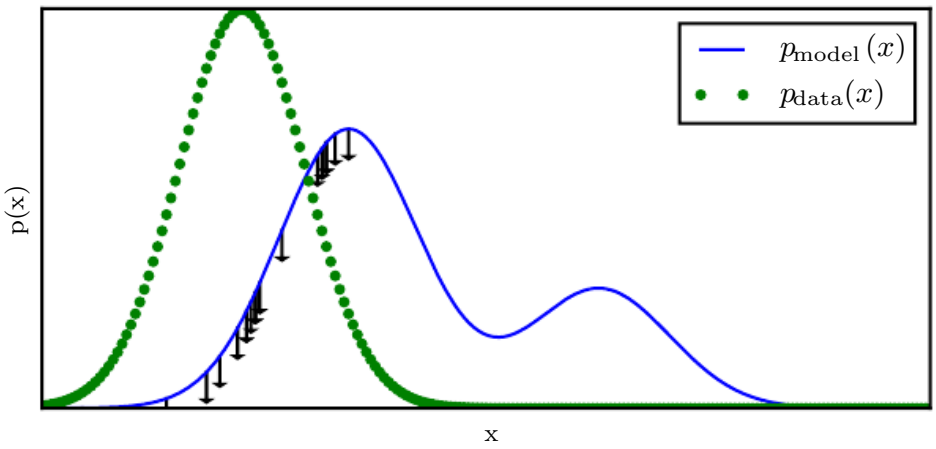

Figure 18.2: An illustration of how the negative phase of contrastive divergence (algorithm 18.2) can fail to suppress spurious modes. A spurious mode is a mode that is present in the model distribution but absent in the data distribution. Because contrastive divergence initializes its Markov chains from data points and runs the Markov chain for only a few steps, it is unlikely to visit modes in the model that are far from the data points. This means that when sampling from the model, we will sometimes get samples that do not resemble the data. It also means that due to wasting some of its probability mass on these modes, the model will struggle to place high probability mass on the correct modes. For the purpose of visualization, this figure uses a somewhat simplified concept of distance—the spurious mode is far from the correct mode along the number line in $\mathbb{R}$. This corresponds to a Markov chain based on making local moves with a single x variable in $\mathbb{R}$. For most deep probabilistic models, the Markov chains are based on Gibbs sampling and can make non-local moves of individual variables but cannot move all of the variables simultaneously. For these problems, it is usually better to consider the edit distance between modes, rather than the Euclidean distance. However, edit distance in a high dimensional space is difficult to depict in a 2-D plot.

Markov chains initialized at training points, unless k is very large.

Carreira-Perpiñan and Hinton (2005) showed experimentally that the CD estimator is biased for RBMs and fully visible Boltzmann machines, in that it converges to different points than the maximum likelihood estimator. They argue that because the bias is small, CD could be used as an inexpensive way to initialize a model that could later be fine-tuned via more expensive MCMC methods. Bengio and Delalleau (2009) showed that CD can be interpreted as discarding the smallest terms of the correct MCMC update gradient, which explains the bias.

CD is useful for training shallow models like RBMs. These can in turn be stacked to initialize deeper models like DBNs or DBMs. However, CD does not provide much help for training deeper models directly. This is because it is difficult

to obtain samples of the hidden units given samples of the visible units. Since the hidden units are not included in the data, initializing from training points cannot solve the problem. Even if we initialize the visible units from the data, we will still need to burn in a Markov chain sampling from the distribution over the hidden units conditioned on those visible samples.

The CD algorithm can be thought of as penalizing the model for having a Markov chain that changes the input rapidly when the input comes from the data. This means training with CD somewhat resembles autoencoder training. Even though CD is more biased than some of the other training methods, it can be useful for pretraining shallow models that will later be stacked. This is because the earliest models in the stack are encouraged to copy more information up to their latent variables, thereby making it available to the later models. This should be thought of more of as an often-exploitable side effect of CD training rather than a principled design advantage.

Sutskever and Tieleman (2010) showed that the CD update direction is not the gradient of any function. This allows for situations where CD could cycle forever, but in practice this is not a serious problem.

A different strategy that resolves many of the problems with CD is to initialize the Markov chains at each gradient step with their states from the previous gradient step. This approach was first discovered under the name **stochastic maximum likelihood** (SML) in the applied mathematics and statistics community (Younes, 1998) and later independently rediscovered under the name **persistent contrastive divergence** (PCD, or PCD-k to indicate the use of k Gibbs steps per update) in the deep learning community (Tieleman, 2008). See algorithm 18.3. The basic idea of this approach is that, so long as the steps taken by the stochastic gradient algorithm are small, then the model from the previous step will be similar to the model from the current step. It follows that the samples from the previous model's distribution will be very close to being fair samples from the current model's distribution, so a Markov chain initialized with these samples will not require much time to mix.

Because each Markov chain is continually updated throughout the learning process, rather than restarted at each gradient step, the chains are free to wander far enough to find all of the model's modes. SML is thus considerably more resistant to forming models with spurious modes than CD is. Moreover, because it is possible to store the state of all of the sampled variables, whether visible or latent, SML provides an initialization point for both the hidden and visible units. CD is only able to provide an initialization for the visible units, and therefore requires burn-in for deep models. SML is able to train deep models efficiently.

Marlin *et al.* (2010) compared SML to many of the other criteria presented in this chapter. They found that SML results in the best test set log-likelihood for an RBM, and that if the RBM's hidden units are used as features for an SVM classifier, SML results in the best classification accuracy.

SML is vulnerable to becoming inaccurate if the stochastic gradient algorithm can move the model faster than the Markov chain can mix between steps. This can happen if k is too small or ϵ is too large. The permissible range of values is unfortunately highly problem-dependent. There is no known way to test formally whether the chain is successfully mixing between steps. Subjectively, if the learning rate is too high for the number of Gibbs steps, the human operator will be able to observe that there is much more variance in the negative phase samples across gradient steps rather than across different Markov chains. For example, a model trained on MNIST might sample exclusively 7s on one step. The learning process will then push down strongly on the mode corresponding to 7s, and the model might sample exclusively 9s on the next step.

Algorithm 18.3 The stochastic maximum likelihood / persistent contrastive divergence algorithm using gradient ascent as the optimization procedure.

Set ϵ, the step size, to a small positive number.
Set k, the number of Gibbs steps, high enough to allow a Markov chain sampling from $p(\mathbf{x}; \boldsymbol{\theta} + \epsilon \mathbf{g})$ to burn in, starting from samples from $p(\mathbf{x}; \boldsymbol{\theta})$. Perhaps 1 for RBM on a small image patch, or 5-50 for a more complicated model like a DBM.
Initialize a set of m samples $\{\tilde{\mathbf{x}}^{(1)}, \ldots, \tilde{\mathbf{x}}^{(m)}\}$ to random values (e.g., from a uniform or normal distribution, or possibly a distribution with marginals matched to the model's marginals).
while not converged **do**
  Sample a minibatch of m examples $\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\}$ from the training set.
  $\mathbf{g} \leftarrow \frac{1}{m} \sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} \log \tilde{p}(\mathbf{x}^{(i)}; \boldsymbol{\theta})$.
  for $i = 1$ to k **do**
    for $j = 1$ to m **do**
      $\tilde{\mathbf{x}}^{(j)} \leftarrow \text{gibbs\_update}(\tilde{\mathbf{x}}^{(j)})$.
    end for
  end for
  $\mathbf{g} \leftarrow \mathbf{g} - \frac{1}{m} \sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} \log \tilde{p}(\tilde{\mathbf{x}}^{(i)}; \boldsymbol{\theta})$.
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \epsilon \mathbf{g}$.
end while

Care must be taken when evaluating the samples from a model trained with SML. It is necessary to draw the samples starting from a fresh Markov chain

initialized from a random starting point after the model is done training. The samples present in the persistent negative chains used for training have been influenced by several recent versions of the model, and thus can make the model appear to have greater capacity than it actually does.

Berglund and Raiko (2013) performed experiments to examine the bias and variance in the estimate of the gradient provided by CD and SML. CD proves to have lower variance than the estimator based on exact sampling. SML has higher variance. The cause of CD's low variance is its use of the same training points in both the positive and negative phase. If the negative phase is initialized from different training points, the variance rises above that of the estimator based on exact sampling.

All of these methods based on using MCMC to draw samples from the model can in principle be used with almost any variant of MCMC. This means that techniques such as SML can be improved by using any of the enhanced MCMC techniques described in chapter 17, such as parallel tempering (Desjardins *et al.*, 2010; Cho *et al.*, 2010).

One approach to accelerating mixing during learning relies not on changing the Monte Carlo sampling technology but rather on changing the parametrization of the model and the cost function. **Fast PCD** or FPCD (Tieleman and Hinton, 2009) involves replacing the parameters $\boldsymbol{\theta}$ of a traditional model with an expression

$$\boldsymbol{\theta} = \boldsymbol{\theta}^{(\text{slow})} + \boldsymbol{\theta}^{(\text{fast})}. \tag{18.16}$$

There are now twice as many parameters as before, and they are added together element-wise to provide the parameters used by the original model definition. The fast copy of the parameters is trained with a much larger learning rate, allowing it to adapt rapidly in response to the negative phase of learning and push the Markov chain to new territory. This forces the Markov chain to mix rapidly, though this effect only occurs during learning while the fast weights are free to change. Typically one also applies significant weight decay to the fast weights, encouraging them to converge to small values, after only transiently taking on large values long enough to encourage the Markov chain to change modes.

One key benefit to the MCMC-based methods described in this section is that they provide an estimate of the gradient of $\log Z$, and thus we can essentially decompose the problem into the $\log \tilde{p}$ contribution and the $\log Z$ contribution. We can then use any other method to tackle $\log \tilde{p}(\mathbf{x})$, and just add our negative phase gradient onto the other method's gradient. In particular, this means that our positive phase can make use of methods that provide only a lower bound on $\tilde{p}$. Most of the other methods of dealing with $\log Z$ presented in this chapter are

incompatible with bound-based positive phase methods.

18.3 Pseudolikelihood

Monte Carlo approximations to the partition function and its gradient directly confront the partition function. Other approaches sidestep the issue, by training the model without computing the partition function. Most of these approaches are based on the observation that it is easy to compute ratios of probabilities in an undirected probabilistic model. This is because the partition function appears in both the numerator and the denominator of the ratio and cancels out:

$$\frac{p(\mathbf{x})}{p(\mathbf{y})} = \frac{\frac{1}{Z}\tilde{p}(\mathbf{x})}{\frac{1}{Z}\tilde{p}(\mathbf{y})} = \frac{\tilde{p}(\mathbf{x})}{\tilde{p}(\mathbf{y})}. \tag{18.17}$$

The pseudolikelihood is based on the observation that conditional probabilities take this ratio-based form, and thus can be computed without knowledge of the partition function. Suppose that we partition $\mathbf{x}$ into $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$, where $\mathbf{a}$ contains the variables we want to find the conditional distribution over, $\mathbf{b}$ contains the variables we want to condition on, and $\mathbf{c}$ contains the variables that are not part of our query.

$$p(\mathbf{a} \mid \mathbf{b}) = \frac{p(\mathbf{a}, \mathbf{b})}{p(\mathbf{b})} = \frac{p(\mathbf{a}, \mathbf{b})}{\sum_{\mathbf{a},\mathbf{c}} p(\mathbf{a}, \mathbf{b}, \mathbf{c})} = \frac{\tilde{p}(\mathbf{a}, \mathbf{b})}{\sum_{\mathbf{a},\mathbf{c}} \tilde{p}(\mathbf{a}, \mathbf{b}, \mathbf{c})}. \tag{18.18}$$

This quantity requires marginalizing out $\mathbf{a}$, which can be a very efficient operation provided that $\mathbf{a}$ and $\mathbf{c}$ do not contain very many variables. In the extreme case, $\mathbf{a}$ can be a single variable and $\mathbf{c}$ can be empty, making this operation require only as many evaluations of $\tilde{p}$ as there are values of a single random variable.

Unfortunately, in order to compute the log-likelihood, we need to marginalize out large sets of variables. If there are n variables total, we must marginalize a set of size $n - 1$. By the chain rule of probability,

$$\log p(\mathbf{x}) = \log p(x_1) + \log p(x_2 \mid x_1) + \cdots + p(x_n \mid \mathbf{x}_{1:n-1}). \tag{18.19}$$

In this case, we have made $\mathbf{a}$ maximally small, but $\mathbf{c}$ can be as large as $\mathbf{x}_{2:n}$. What if we simply move $\mathbf{c}$ into $\mathbf{b}$ to reduce the computational cost? This yields the **pseudolikelihood** (Besag, 1975) objective function, based on predicting the value of feature x_i given all of the other features $\boldsymbol{x}_{-i}$:

$$\sum_{i=1}^{n} \log p(x_i \mid \boldsymbol{x}_{-i}). \tag{18.20}$$

If each random variable has k different values, this requires only $k \times n$ evaluations of $\tilde{p}$ to compute, as opposed to the k^n evaluations needed to compute the partition function.

This may look like an unprincipled hack, but it can be proven that estimation by maximizing the pseudolikelihood is asymptotically consistent (Mase, 1995). Of course, in the case of datasets that do not approach the large sample limit, pseudolikelihood may display different behavior from the maximum likelihood estimator.

It is possible to trade computational complexity for deviation from maximum likelihood behavior by using the **generalized pseudolikelihood** estimator (Huang and Ogata, 2002). The generalized pseudolikelihood estimator uses m different sets $\mathbb{S}^{(i)}, i = 1, \ldots, m$ of indices of variables that appear together on the left side of the conditioning bar. In the extreme case of $m = 1$ and $\mathbb{S}^{(1)} = 1, \ldots, n$ the generalized pseudolikelihood recovers the log-likelihood. In the extreme case of $m = n$ and $\mathbb{S}^{(i)} = \{i\}$, the generalized pseudolikelihood recovers the pseudolikelihood. The generalized pseudolikelihood objective function is given by

$$\sum_{i=1}^{m} \log p(\mathbf{x}_{\mathbb{S}^{(i)}} \mid \mathbf{x}_{-\mathbb{S}^{(i)}}). \tag{18.21}$$

The performance of pseudolikelihood-based approaches depends largely on how the model will be used. Pseudolikelihood tends to perform poorly on tasks that require a good model of the full joint $p(\mathbf{x})$, such as density estimation and sampling. However, it can perform better than maximum likelihood for tasks that require only the conditional distributions used during training, such as filling in small amounts of missing values. Generalized pseudolikelihood techniques are especially powerful if the data has regular structure that allows the $\mathbb{S}$ index sets to be designed to capture the most important correlations while leaving out groups of variables that only have negligible correlation. For example, in natural images, pixels that are widely separated in space also have weak correlation, so the generalized pseudolikelihood can be applied with each $\mathbb{S}$ set being a small, spatially localized window.

One weakness of the pseudolikelihood estimator is that it cannot be used with other approximations that provide only a lower bound on $\tilde{p}(\mathbf{x})$, such as variational inference, which will be covered in chapter 19. This is because $\tilde{p}$ appears in the denominator. A lower bound on the denominator provides only an upper bound on the expression as a whole, and there is no benefit to maximizing an upper bound. This makes it difficult to apply pseudolikelihood approaches to deep models such as deep Boltzmann machines, since variational methods are one of the dominant approaches to approximately marginalizing out the many layers of hidden variables

that interact with each other. However, pseudolikelihood is still useful for deep learning, because it can be used to train single layer models, or deep models using approximate inference methods that are not based on lower bounds.

Pseudolikelihood has a much greater cost per gradient step than SML, due to its explicit computation of all of the conditionals. However, generalized pseudolikelihood and similar criteria can still perform well if only one randomly selected conditional is computed per example (Goodfellow *et al.*, 2013b), thereby bringing the computational cost down to match that of SML.

Though the pseudolikelihood estimator does not explicitly minimize $\log Z$, it can still be thought of as having something resembling a negative phase. The denominators of each conditional distribution result in the learning algorithm suppressing the probability of all states that have only one variable differing from a training example.

See Marlin and de Freitas (2011) for a theoretical analysis of the asymptotic efficiency of pseudolikelihood.

18.4 Score Matching and Ratio Matching

Score matching (Hyvärinen, 2005) provides another consistent means of training a model without estimating Z or its derivatives. The name score matching comes from terminology in which the derivatives of a log density with respect to its argument, $\nabla_{\boldsymbol{x}} \log p(\boldsymbol{x})$, are called its **score**. The strategy used by score matching is to minimize the expected squared difference between the derivatives of the model's log density with respect to the input and the derivatives of the data's log density with respect to the input:

$$L(\boldsymbol{x}, \boldsymbol{\theta}) = \frac{1}{2} ||\nabla_{\boldsymbol{x}} \log p_{\text{model}}(\boldsymbol{x}; \boldsymbol{\theta}) - \nabla_{\boldsymbol{x}} \log p_{\text{data}}(\boldsymbol{x})||_2^2 \tag{18.22}$$

$$J(\boldsymbol{\theta}) = \frac{1}{2} \mathbb{E}_{p_{\text{data}}(\boldsymbol{x})} L(\boldsymbol{x}, \boldsymbol{\theta}) \tag{18.23}$$

$$\boldsymbol{\theta}^* = \min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \tag{18.24}$$

This objective function avoids the difficulties associated with differentiating the partition function Z because Z is not a function of $\boldsymbol{x}$ and therefore $\nabla_{\mathbf{x}} Z = 0$. Initially, score matching appears to have a new difficulty: computing the score of the data distribution requires knowledge of the true distribution generating the training data, p_{data}. Fortunately, minimizing the expected value of $L(\boldsymbol{x}, \boldsymbol{\theta})$ is

equivalent to minimizing the expected value of

$$\tilde{L}(\boldsymbol{x}, \boldsymbol{\theta}) = \sum_{j=1}^{n} \left(\frac{\partial^2}{\partial x_j^2} \log p_{\text{model}}(\boldsymbol{x}; \boldsymbol{\theta}) + \frac{1}{2} \left(\frac{\partial}{\partial x_j} \log p_{\text{model}}(\boldsymbol{x}; \boldsymbol{\theta}) \right)^2 \right) \tag{18.25}$$

where n is the dimensionality of $\boldsymbol{x}$.

Because score matching requires taking derivatives with respect to $\mathbf{x}$, it is not applicable to models of discrete data. However, the latent variables in the model may be discrete.

Like the pseudolikelihood, score matching only works when we are able to evaluate $\log \tilde{p}(\mathbf{x})$ and its derivatives directly. It is not compatible with methods that only provide a lower bound on $\log \tilde{p}(\mathbf{x})$, because score matching requires the derivatives and second derivatives of $\log \tilde{p}(\mathbf{x})$ and a lower bound conveys no information about its derivatives. This means that score matching cannot be applied to estimating models with complicated interactions between the hidden units, such as sparse coding models or deep Boltzmann machines. While score matching can be used to pretrain the first hidden layer of a larger model, it has not been applied as a pretraining strategy for the deeper layers of a larger model. This is probably because the hidden layers of such models usually contain some discrete variables.

While score matching does not explicitly have a negative phase, it can be viewed as a version of contrastive divergence using a specific kind of Markov chain (Hyvärinen, 2007a). The Markov chain in this case is not Gibbs sampling, but rather a different approach that makes local moves guided by the gradient. Score matching is equivalent to CD with this type of Markov chain when the size of the local moves approaches zero.

Lyu (2009) generalized score matching to the discrete case (but made an error in their derivation that was corrected by Marlin *et al.* (2010)). Marlin *et al.* (2010) found that **generalized score matching** (GSM) does not work in high dimensional discrete spaces where the observed probability of many events is 0.

A more successful approach to extending the basic ideas of score matching to discrete data is **ratio matching** (Hyvärinen, 2007b). Ratio matching applies specifically to binary data. Ratio matching consists of minimizing the average over examples of the following objective function:

$$L^{(\text{RM})}(\boldsymbol{x}, \boldsymbol{\theta}) = \sum_{j=1}^{n} \left(\frac{1}{1 + \frac{p_{\text{model}}(\boldsymbol{x}; \boldsymbol{\theta})}{p_{\text{model}}(f(\boldsymbol{x}, j); \boldsymbol{\theta})}} \right)^2, \tag{18.26}$$

where $f(\boldsymbol{x}, j)$ returns $\mathbf{x}$ with the bit at position j flipped. Ratio matching avoids the partition function using the same trick as the pseudolikelihood estimator: in a ratio of two probabilities, the partition function cancels out. Marlin *et al.* (2010) found that ratio matching outperforms SML, pseudolikelihood and GSM in terms of the ability of models trained with ratio matching to denoise test set images.

Like the pseudolikelihood estimator, ratio matching requires n evaluations of $\tilde{p}$ per data point, making its computational cost per update roughly n times higher than that of SML.

As with the pseudolikelihood estimator, ratio matching can be thought of as pushing down on all fantasy states that have only one variable different from a training example. Since ratio matching applies specifically to binary data, this means that it acts on all fantasy states within Hamming distance 1 of the data.

Ratio matching can also be useful as the basis for dealing with high-dimensional sparse data, such as word count vectors. This kind of data poses a challenge for MCMC-based methods because the data is extremely expensive to represent in dense format, yet the MCMC sampler does not yield sparse values until the model has learned to represent the sparsity in the data distribution. Dauphin and Bengio (2013) overcame this issue by designing an unbiased stochastic approximation to ratio matching. The approximation evaluates only a randomly selected subset of the terms of the objective, and does not require the model to generate complete fantasy samples.

See Marlin and de Freitas (2011) for a theoretical analysis of the asymptotic efficiency of ratio matching.

18.5 Denoising Score Matching

In some cases we may wish to regularize score matching, by fitting a distribution

$$p_{\text{smoothed}}(\boldsymbol{x}) = \int p_{\text{data}}(\boldsymbol{y}) q(\boldsymbol{x} \mid \boldsymbol{y}) d\boldsymbol{y} \tag{18.27}$$

rather than the true p_{data}. The distribution $q(\boldsymbol{x} \mid \boldsymbol{y})$ is a corruption process, usually one that forms $\boldsymbol{x}$ by adding a small amount of noise to $\boldsymbol{y}$.

Denoising score matching is especially useful because in practice we usually do not have access to the true p_{data} but rather only an empirical distribution defined by samples from it. Any consistent estimator will, given enough capacity, make p_{model} into a set of Dirac distributions centered on the training points. Smoothing by q helps to reduce this problem, at the loss of the asymptotic consistency property

described in section 5.4.5. Kingma and LeCun (2010) introduced a procedure for performing regularized score matching with the smoothing distribution q being normally distributed noise.

Recall from section 14.5.1 that several autoencoder training algorithms are equivalent to score matching or denoising score matching. These autoencoder training algorithms are therefore a way of overcoming the partition function problem.

18.6 Noise-Contrastive Estimation

Most techniques for estimating models with intractable partition functions do not provide an estimate of the partition function. SML and CD estimate only the gradient of the log partition function, rather than the partition function itself. Score matching and pseudolikelihood avoid computing quantities related to the partition function altogether.

Noise-contrastive estimation (NCE) (Gutmann and Hyvarinen, 2010) takes a different strategy. In this approach, the probability distribution estimated by the model is represented explicitly as

$$\log p_{\text{model}}(\mathbf{x}) = \log \tilde{p}_{\text{model}}(\mathbf{x}; \boldsymbol{\theta}) + c, \tag{18.28}$$

where c is explicitly introduced as an approximation of $-\log Z(\boldsymbol{\theta})$. Rather than estimating only $\boldsymbol{\theta}$, the noise contrastive estimation procedure treats c as just another parameter and estimates $\boldsymbol{\theta}$ and c simultaneously, using the same algorithm for both. The resulting $\log p_{\text{model}}(\mathbf{x})$ thus may not correspond exactly to a valid probability distribution, but will become closer and closer to being valid as the estimate of c improves.[1]

Such an approach would not be possible using maximum likelihood as the criterion for the estimator. The maximum likelihood criterion would choose to set c arbitrarily high, rather than setting c to create a valid probability distribution.

NCE works by reducing the unsupervised learning problem of estimating $p(\mathbf{x})$ to that of learning a probabilistic binary classifier in which one of the categories corresponds to the data generated by the model. This supervised learning problem is constructed in such a way that maximum likelihood estimation in this supervised

[1]NCE is also applicable to problems with a tractable partition function, where there is no need to introduce the extra parameter c. However, it has generated the most interest as a means of estimating models with difficult partition functions.

learning problem defines an asymptotically consistent estimator of the original problem.

Specifically, we introduce a second distribution, the **noise distribution** $p_{\text{noise}}(\mathbf{x})$. The noise distribution should be tractable to evaluate and to sample from. We can now construct a model over both $\mathbf{x}$ and a new, binary class variable y. In the new joint model, we specify that

$$p_{\text{joint}}(y = 1) = \frac{1}{2}, \tag{18.29}$$

$$p_{\text{joint}}(\mathbf{x} \mid y = 1) = p_{\text{model}}(\mathbf{x}), \tag{18.30}$$

and

$$p_{\text{joint}}(\mathbf{x} \mid y = 0) = p_{\text{noise}}(\mathbf{x}). \tag{18.31}$$

In other words, y is a switch variable that determines whether we will generate $\mathbf{x}$ from the model or from the noise distribution.

We can construct a similar joint model of training data. In this case, the switch variable determines whether we draw $\mathbf{x}$ from the **data** or from the noise distribution. Formally, $p_{\text{train}}(y = 1) = \frac{1}{2}$, $p_{\text{train}}(\mathbf{x} \mid y = 1) = p_{\text{data}}(\mathbf{x})$, and $p_{\text{train}}(\mathbf{x} \mid y = 0) = p_{\text{noise}}(\mathbf{x})$.

We can now just use standard maximum likelihood learning on the **supervised** learning problem of fitting p_{joint} to p_{train}:

$$\boldsymbol{\theta}, c = \arg\max_{\boldsymbol{\theta}, c} \mathbb{E}_{\mathbf{x},\text{y} \sim p_{\text{train}}} \log p_{\text{joint}}\left(y \mid \mathbf{x}\right). \tag{18.32}$$

The distribution p_{joint} is essentially a logistic regression model applied to the difference in log probabilities of the model and the noise distribution:

$$p_{\text{joint}}\left(y = 1 \mid \mathbf{x}\right) = \frac{p_{\text{model}}\left(\mathbf{x}\right)}{p_{\text{model}}(\mathbf{x}) + p_{\text{noise}}(\mathbf{x})} \tag{18.33}$$

$$= \frac{1}{1 + \frac{p_{\text{noise}}(\mathbf{x})}{p_{\text{model}}\left(\mathbf{x}\right)}} \tag{18.34}$$

$$= \frac{1}{1 + \exp\left(\log \frac{p_{\text{noise}}(\mathbf{x})}{p_{\text{model}}\left(\mathbf{x}\right)}\right)} \tag{18.35}$$

$$= \sigma\left(-\log \frac{p_{\text{noise}}(\mathbf{x})}{p_{\text{model}}(\mathbf{x})}\right) \tag{18.36}$$

$$= \sigma\left(\log p_{\text{model}}(\mathbf{x}) - \log p_{\text{noise}}(\mathbf{x})\right). \tag{18.37}$$

NCE is thus simple to apply so long as $\log \tilde{p}_{\text{model}}$ is easy to back-propagate through, and, as specified above, p_{noise} is easy to evaluate (in order to evaluate p_{joint}) and sample from (in order to generate the training data).

NCE is most successful when applied to problems with few random variables, but can work well even if those random variables can take on a high number of values. For example, it has been successfully applied to modeling the conditional distribution over a word given the context of the word (Mnih and Kavukcuoglu, 2013). Though the word may be drawn from a large vocabulary, there is only one word.

When NCE is applied to problems with many random variables, it becomes less efficient. The logistic regression classifier can reject a noise sample by identifying any one variable whose value is unlikely. This means that learning slows down greatly after p_{model} has learned the basic marginal statistics. Imagine learning a model of images of faces, using unstructured Gaussian noise as p_{noise}. If p_{model} learns about eyes, it can reject almost all unstructured noise samples without having learned anything about other facial features, such as mouths.

The constraint that p_{noise} must be easy to evaluate and easy to sample from can be overly restrictive. When p_{noise} is simple, most samples are likely to be too obviously distinct from the data to force p_{model} to improve noticeably.

Like score matching and pseudolikelihood, NCE does not work if only a lower bound on $\tilde{p}$ is available. Such a lower bound could be used to construct a lower bound on $p_{\text{joint}}(y = 1 \mid \mathbf{x})$, but it can only be used to construct an upper bound on $p_{\text{joint}}(y = 0 \mid \mathbf{x})$, which appears in half the terms of the NCE objective. Likewise, a lower bound on p_{noise} is not useful, because it provides only an upper bound on $p_{\text{joint}}(y = 1 \mid \mathbf{x})$.

When the model distribution is copied to define a new noise distribution before each gradient step, NCE defines a procedure called **self-contrastive estimation**, whose expected gradient is equivalent to the expected gradient of maximum likelihood (Goodfellow, 2014). The special case of NCE where the noise samples are those generated by the model suggests that maximum likelihood can be interpreted as a procedure that forces a model to constantly learn to distinguish reality from its own evolving beliefs, while noise contrastive estimation achieves some reduced computational cost by only forcing the model to distinguish reality from a fixed baseline (the noise model).

Using the supervised task of classifying between training samples and generated samples (with the model energy function used in defining the classifier) to provide a gradient on the model was introduced earlier in various forms (Welling *et al.*, 2003b; Bengio, 2009).

Noise contrastive estimation is based on the idea that a good generative model should be able to distinguish data from noise. A closely related idea is that a good generative model should be able to generate samples that no classifier can distinguish from data. This idea yields generative adversarial networks (section 20.10.4).

18.7 Estimating the Partition Function

While much of this chapter is dedicated to describing methods that avoid needing to compute the intractable partition function $Z(\boldsymbol{\theta})$ associated with an undirected graphical model, in this section we discuss several methods for directly estimating the partition function.

Estimating the partition function can be important because we require it if we wish to compute the normalized likelihood of data. This is often important in *evaluating* the model, monitoring training performance, and comparing models to each other.

For example, imagine we have two models: model $\mathcal{M}_A$ defining a probability distribution $p_A(\mathbf{x};\boldsymbol{\theta}_A) = \frac{1}{Z_A}\tilde{p}_A(\mathbf{x};\boldsymbol{\theta}_A)$ and model $\mathcal{M}_B$ defining a probability distribution $p_B(\mathbf{x};\boldsymbol{\theta}_B) = \frac{1}{Z_B}\tilde{p}_B(\mathbf{x};\boldsymbol{\theta}_B)$. A common way to compare the models is to evaluate and compare the likelihood that both models assign to an i.i.d. test dataset. Suppose the test set consists of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$. If $\prod_i p_A(\mathrm{x}^{(i)};\boldsymbol{\theta}_A) > \prod_i p_B(\mathrm{x}^{(i)};\boldsymbol{\theta}_B)$ or equivalently if

$$\sum_i \log p_A(\mathrm{x}^{(i)};\boldsymbol{\theta}_A) - \sum_i \log p_B(\mathrm{x}^{(i)};\boldsymbol{\theta}_B) > 0, \tag{18.38}$$

then we say that $\mathcal{M}_A$ is a better model than $\mathcal{M}_B$ (or, at least, it is a better model of the test set), in the sense that it has a better test log-likelihood. Unfortunately, testing whether this condition holds requires knowledge of the partition function. Unfortunately, equation 18.38 seems to require evaluating the log probability that the model assigns to each point, which in turn requires evaluating the partition function. We can simplify the situation slightly by re-arranging equation 18.38 into a form where we need to know only the **ratio** of the two model's partition functions:

$$\sum_i \log p_A(\mathbf{x}^{(i)};\boldsymbol{\theta}_A) - \sum_i \log p_B(\mathbf{x}^{(i)};\boldsymbol{\theta}_B) = \sum_i \left(\log \frac{\tilde{p}_A(\mathbf{x}^{(i)};\boldsymbol{\theta}_A)}{\tilde{p}_B(\mathbf{x}^{(i)};\boldsymbol{\theta}_B)} \right) - m \log \frac{Z(\boldsymbol{\theta}_A)}{Z(\boldsymbol{\theta}_B)}. \tag{18.39}$$

We can thus determine whether $\mathcal{M}_A$ is a better model than $\mathcal{M}_B$ without knowing the partition function of either model but only their ratio. As we will see shortly, we can estimate this ratio using importance sampling, provided that the two models are similar.

If, however, we wanted to compute the actual probability of the test data under either $\mathcal{M}_A$ or $\mathcal{M}_B$, we would need to compute the actual value of the partition functions. That said, if we knew the ratio of two partition functions, $r = \frac{Z(\boldsymbol{\theta}_B)}{Z(\boldsymbol{\theta}_A)}$, and we knew the actual value of just one of the two, say $Z(\boldsymbol{\theta}_A)$, we could compute the value of the other:

$$Z(\boldsymbol{\theta}_B) = rZ(\boldsymbol{\theta}_A) = \frac{Z(\boldsymbol{\theta}_B)}{Z(\boldsymbol{\theta}_A)} Z(\boldsymbol{\theta}_A). \tag{18.40}$$

A simple way to estimate the partition function is to use a Monte Carlo method such as simple importance sampling. We present the approach in terms of continuous variables using integrals, but it can be readily applied to discrete variables by replacing the integrals with summation. We use a proposal distribution $p_0(\mathbf{x}) = \frac{1}{Z_0}\tilde{p}_0(\mathbf{x})$ which supports tractable sampling and tractable evaluation of both the partition function Z_0 and the unnormalized distribution $\tilde{p}_0(\mathbf{x})$.

$$Z_1 = \int \tilde{p}_1(\mathbf{x})\ d\mathbf{x} \tag{18.41}$$

$$= \int \frac{p_0(\mathbf{x})}{p_0(\mathbf{x})} \tilde{p}_1(\mathbf{x})\ d\mathbf{x} \tag{18.42}$$

$$= Z_0 \int p_0(\mathbf{x}) \frac{\tilde{p}_1(\mathbf{x})}{\tilde{p}_0(\mathbf{x})}\ d\mathbf{x} \tag{18.43}$$

$$\hat{Z}_1 = \frac{Z_0}{K} \sum_{k=1}^{K} \frac{\tilde{p}_1(\mathbf{x}^{(k)})}{\tilde{p}_0(\mathbf{x}^{(k)})} \quad \text{s.t.} : \mathbf{x}^{(k)} \sim p_0 \tag{18.44}$$

In the last line, we make a Monte Carlo estimator, $\hat{Z}_1$, of the integral using samples drawn from $p_0(\mathbf{x})$ and then weight each sample with the ratio of the unnormalized $\tilde{p}_1$ and the proposal p_0.

We see also that this approach allows us to estimate the ratio between the partition functions as

$$\frac{1}{K} \sum_{k=1}^{K} \frac{\tilde{p}_1(\mathbf{x}^{(k)})}{\tilde{p}_0(\mathbf{x}^{(k)})} \quad \text{s.t.} : \mathbf{x}^{(k)} \sim p_0. \tag{18.45}$$

This value can then be used directly to compare two models as described in equation 18.39.

If the distribution p_0 is close to p_1, equation 18.44 can be an effective way of estimating the partition function (Minka, 2005). Unfortunately, most of the time p_1 is both complicated (usually multimodal) and defined over a high dimensional space. It is difficult to find a tractable p_0 that is simple enough to evaluate while still being close enough to p_1 to result in a high quality approximation. If p_0 and p_1 are not close, most samples from p_0 will have low probability under p_1 and therefore make (relatively) negligible contribution to the sum in equation 18.44.

Having few samples with significant weights in this sum will result in an estimator that is of poor quality due to high variance. This can be understood quantitatively through an estimate of the variance of our estimate $\hat{Z}_1$:

$$\hat{\text{Var}}\left(\hat{Z}_1\right) = \frac{Z_0}{K^2}\sum_{k=1}^{K}\left(\frac{\tilde{p}_1(\mathbf{x}^{(k)})}{\tilde{p}_0(\mathbf{x}^{(k)})} - \hat{Z}_1\right)^2. \tag{18.46}$$

This quantity is largest when there is significant deviation in the values of the importance weights $\frac{\tilde{p}_1(\mathbf{x}^{(k)})}{\tilde{p}_0(\mathbf{x}^{(k)})}$.

We now turn to two related strategies developed to cope with the challenging task of estimating partition functions for complex distributions over high-dimensional spaces: annealed importance sampling and bridge sampling. Both start with the simple importance sampling strategy introduced above and both attempt to overcome the problem of the proposal p_0 being too far from p_1 by introducing intermediate distributions that attempt to *bridge the gap* between p_0 and p_1.

18.7.1 Annealed Importance Sampling

In situations where $D_{\text{KL}}(p_0\|p_1)$ is large (i.e., where there is little overlap between p_0 and p_1), a strategy called **annealed importance sampling** (AIS) attempts to bridge the gap by introducing intermediate distributions (Jarzynski, 1997; Neal, 2001). Consider a sequence of distributions $p_{\eta_0}, \ldots, p_{\eta_n}$, with $0 = \eta_0 < \eta_1 < \cdots < \eta_{n-1} < \eta_n = 1$ so that the first and last distributions in the sequence are p_0 and p_1 respectively.

This approach allows us to estimate the partition function of a multimodal distribution defined over a high-dimensional space (such as the distribution defined by a trained RBM). We begin with a simpler model with a known partition function (such as an RBM with zeroes for weights) and estimate the ratio between the two model's partition functions. The estimate of this ratio is based on the estimate of the ratios of a sequence of many similar distributions, such as the sequence of RBMs with weights interpolating between zero and the learned weights.

We can now write the ratio $\frac{Z_1}{Z_0}$ as

$$\frac{Z_1}{Z_0} = \frac{Z_1}{Z_0}\frac{Z_{\eta_1}}{Z_{\eta_1}} \cdots \frac{Z_{\eta_{n-1}}}{Z_{\eta_{n-1}}} \tag{18.47}$$

$$= \frac{Z_{\eta_1}}{Z_0}\frac{Z_{\eta_2}}{Z_{\eta_1}} \cdots \frac{Z_{\eta_{n-1}}}{Z_{\eta_{n-2}}}\frac{Z_1}{Z_{\eta_{n-1}}} \tag{18.48}$$

$$= \prod_{j=0}^{n-1} \frac{Z_{\eta_{j+1}}}{Z_{\eta_j}} \tag{18.49}$$

Provided the distributions p_{η_j} and p_{η_j+1}, for all $0 \leq j \leq n-1$, are sufficiently close, we can reliably estimate each of the factors $\frac{Z_{\eta_j+1}}{Z_{\eta_j}}$ using simple importance sampling and then use these to obtain an estimate of $\frac{Z_1}{Z_0}$.

Where do these intermediate distributions come from? Just as the original proposal distribution p_0 is a design choice, so is the sequence of distributions $p_{\eta_1} \ldots p_{\eta_{n-1}}$. That is, it can be specifically constructed to suit the problem domain. One general-purpose and popular choice for the intermediate distributions is to use the weighted geometric average of the target distribution p_1 and the starting proposal distribution (for which the partition function is known) p_0:

$$p_{\eta_j} \propto p_1^{\eta_j} p_0^{1-\eta_j} \tag{18.50}$$

In order to sample from these intermediate distributions, we define a series of Markov chain transition functions $T_{\eta_j}(\boldsymbol{x}' \mid \boldsymbol{x})$ that define the conditional probability distribution of transitioning to $\boldsymbol{x}'$ given we are currently at $\boldsymbol{x}$. The transition operator $T_{\eta_j}(\boldsymbol{x}' \mid \boldsymbol{x})$ is defined to leave $p_{\eta_j}(\boldsymbol{x})$ invariant:

$$p_{\eta_j}(\boldsymbol{x}) = \int p_{\eta_j}(\boldsymbol{x}')T_{\eta_j}(\boldsymbol{x} \mid \boldsymbol{x}')\ d\boldsymbol{x}' \tag{18.51}$$

These transitions may be constructed as any Markov chain Monte Carlo method (e.g., Metropolis-Hastings, Gibbs), including methods involving multiple passes through all of the random variables or other kinds of iterations.

The AIS sampling strategy is then to generate samples from p_0 and then use the transition operators to sequentially generate samples from the intermediate distributions until we arrive at samples from the target distribution p_1:

- for $k = 1 \ldots K$
  - Sample $\boldsymbol{x}_{\eta_1}^{(k)} \sim p_0(\mathbf{x})$

- Sample $\boldsymbol{x}_{\eta_2}^{(k)} \sim T_{\eta_1}(\mathbf{x}_{\eta_2}^{(k)} \mid \boldsymbol{x}_{\eta_1}^{(k)})$
- ...
- Sample $\boldsymbol{x}_{\eta_{n-1}}^{(k)} \sim T_{\eta_{n-2}}(\mathbf{x}_{\eta_{n-1}}^{(k)} \mid \boldsymbol{x}_{\eta_{n-2}}^{(k)})$
- Sample $\boldsymbol{x}_{\eta_n}^{(k)} \sim T_{\eta_{n-1}}(\mathbf{x}_{\eta_n}^{(k)} \mid \boldsymbol{x}_{\eta_{n-1}}^{(k)})$

- end

For sample k, we can derive the importance weight by chaining together the importance weights for the jumps between the intermediate distributions given in equation 18.49:

$$w^{(k)} = \frac{\tilde{p}_{\eta_1}(\boldsymbol{x}_{\eta_1}^{(k)})}{\tilde{p}_0(\boldsymbol{x}_{\eta_1}^{(k)})} \frac{\tilde{p}_{\eta_2}(\boldsymbol{x}_{\eta_2}^{(k)})}{\tilde{p}_{\eta_1}(\boldsymbol{x}_{\eta_2}^{(k)})} \cdots \frac{\tilde{p}_1(\boldsymbol{x}_1^{(k)})}{\tilde{p}_{\eta_{n-1}}(\boldsymbol{x}_{\eta_n}^{(k)})}. \tag{18.52}$$

To avoid numerical issues such as overflow, it is probably best to compute $\log w^{(k)}$ by adding and subtracting log probabilities, rather than computing $w^{(k)}$ by multiplying and dividing probabilities.

With the sampling procedure thus defined and the importance weights given in equation 18.52, the estimate of the ratio of partition functions is given by:

$$\frac{Z_1}{Z_0} \approx \frac{1}{K} \sum_{k=1}^{K} w^{(k)} \tag{18.53}$$

In order to verify that this procedure defines a valid importance sampling scheme, we can show (Neal, 2001) that the AIS procedure corresponds to simple importance sampling on an extended state space with points sampled over the product space $[\boldsymbol{x}_{\eta_1}, \ldots, \boldsymbol{x}_{\eta_{n-1}}, \boldsymbol{x}_1]$. To do this, we define the distribution over the extended space as:

$$\tilde{p}(\boldsymbol{x}_{\eta_1}, \ldots, \boldsymbol{x}_{\eta_{n-1}}, \boldsymbol{x}_1) \tag{18.54}$$

$$= \tilde{p}_1(\boldsymbol{x}_1) \tilde{T}_{\eta_{n-1}}(\boldsymbol{x}_{\eta_{n-1}} \mid \boldsymbol{x}_1) \tilde{T}_{\eta_{n-2}}(\boldsymbol{x}_{\eta_{n-2}} \mid \boldsymbol{x}_{\eta_{n-1}}) \ldots \tilde{T}_{\eta_1}(\boldsymbol{x}_{\eta_1} \mid \boldsymbol{x}_{\eta_2}), \tag{18.55}$$

where $\tilde{T}_a$ is the reverse of the transition operator defined by T_a (via an application of Bayes' rule):

$$\tilde{T}_a(\boldsymbol{x}' \mid \boldsymbol{x}) = \frac{p_a(\boldsymbol{x}')}{p_a(\boldsymbol{x})} T_a(\boldsymbol{x} \mid \boldsymbol{x}') = \frac{\tilde{p}_a(\boldsymbol{x}')}{\tilde{p}_a(\boldsymbol{x})} T_a(\boldsymbol{x} \mid \boldsymbol{x}'). \tag{18.56}$$

Plugging the above into the expression for the joint distribution on the extended state space given in equation 18.55, we get:

$$\tilde{p}(\boldsymbol{x}_{\eta_1}, \ldots, \boldsymbol{x}_{\eta_{n-1}}, \boldsymbol{x}_1) \tag{18.57}$$

$$= \tilde{p}_1(\boldsymbol{x}_1)\frac{\tilde{p}_{\eta_{n-1}}(\boldsymbol{x}_{\eta_{n-1}})}{\tilde{p}_{\eta_{n-1}}(\boldsymbol{x}_1)}T_{\eta_{n-1}}(\boldsymbol{x}_1 \mid \boldsymbol{x}_{\eta_{n-1}})\prod_{i=1}^{n-2}\frac{\tilde{p}_{\eta_i}(\boldsymbol{x}_{\eta_i})}{\tilde{p}_{\eta_i}(\boldsymbol{x}_{\eta_{i+1}})}T_{\eta_i}(\boldsymbol{x}_{\eta_{i+1}} \mid \boldsymbol{x}_{\eta_i}) \tag{18.58}$$

$$= \frac{\tilde{p}_1(\boldsymbol{x}_1)}{\tilde{p}_{\eta_{n-1}}(\boldsymbol{x}_1)}T_{\eta_{n-1}}(\boldsymbol{x}_1 \mid \boldsymbol{x}_{\eta_{n-1}})\,\tilde{p}_{\eta_1}(\boldsymbol{x}_{\eta_1})\prod_{i=1}^{n-2}\frac{\tilde{p}_{\eta_{i+1}}(\boldsymbol{x}_{\eta_{i+1}})}{\tilde{p}_{\eta_i}(\boldsymbol{x}_{\eta_{i+1}})}T_{\eta_i}(\boldsymbol{x}_{\eta_{i+1}} \mid \boldsymbol{x}_{\eta_i}). \tag{18.59}$$

We now have means of generating samples from the joint proposal distribution q over the extended sample via a sampling scheme given above, with the joint distribution given by:

$$q(\boldsymbol{x}_{\eta_1}, \ldots, \boldsymbol{x}_{\eta_{n-1}}, \boldsymbol{x}_1) = p_0(\boldsymbol{x}_{\eta_1})T_{\eta_1}(\boldsymbol{x}_{\eta_2} \mid \boldsymbol{x}_{\eta_1}) \ldots T_{\eta_{n-1}}(\boldsymbol{x}_1 \mid \boldsymbol{x}_{\eta_{n-1}}). \tag{18.60}$$

We have a joint distribution on the extended space given by equation 18.59. Taking $q(\boldsymbol{x}_{\eta_1}, \ldots, \boldsymbol{x}_{\eta_{n-1}}, \boldsymbol{x}_1)$ as the proposal distribution on the extended state space from which we will draw samples, it remains to determine the importance weights:

$$w^{(k)} = \frac{\tilde{p}(\boldsymbol{x}_{\eta_1}, \ldots, \boldsymbol{x}_{\eta_{n-1}}, \boldsymbol{x}_1)}{q(\boldsymbol{x}_{\eta_1}, \ldots, \boldsymbol{x}_{\eta_{n-1}}, \boldsymbol{x}_1)} = \frac{\tilde{p}_1(\boldsymbol{x}_1^{(k)})}{\tilde{p}_{\eta_{n-1}}(\boldsymbol{x}_{\eta_{n-1}}^{(k)})} \ldots \frac{\tilde{p}_{\eta_2}(\boldsymbol{x}_{\eta_2}^{(k)})}{\tilde{p}_1(\boldsymbol{x}_{\eta_1}^{(k)})}\frac{\tilde{p}_{\eta_1}(\boldsymbol{x}_{\eta_1}^{(k)})}{\tilde{p}_0(\boldsymbol{x}_0^{(k)})}. \tag{18.61}$$

These weights are the same as proposed for AIS. Thus we can interpret AIS as simple importance sampling applied to an extended state and its validity follows immediately from the validity of importance sampling.

Annealed importance sampling (AIS) was first discovered by Jarzynski (1997) and then again, independently, by Neal (2001). It is currently the most common way of estimating the partition function for undirected probabilistic models. The reasons for this may have more to do with the publication of an influential paper (Salakhutdinov and Murray, 2008) describing its application to estimating the partition function of restricted Boltzmann machines and deep belief networks than with any inherent advantage the method has over the other method described below.

A discussion of the properties of the AIS estimator (e.g.. its variance and efficiency) can be found in Neal (2001).

18.7.2 Bridge Sampling

Bridge sampling Bennett (1976) is another method that, like AIS, addresses the shortcomings of importance sampling. Rather than chaining together a series of

intermediate distributions, bridge sampling relies on a single distribution p_*, known as the bridge, to interpolate between a distribution with known partition function, p_0, and a distribution p_1 for which we are trying to estimate the partition function Z_1.

Bridge sampling estimates the ratio Z_1/Z_0 as the ratio of the expected importance weights between $\tilde{p}_0$ and $\tilde{p}_*$ and between $\tilde{p}_1$ and $\tilde{p}_*$:

$$\frac{Z_1}{Z_0} \approx \sum_{k=1}^{K} \frac{\tilde{p}_*(\boldsymbol{x}_0^{(k)})}{\tilde{p}_0(\boldsymbol{x}_0^{(k)})} \Bigg/ \sum_{k=1}^{K} \frac{\tilde{p}_*(\boldsymbol{x}_1^{(k)})}{\tilde{p}_1(\boldsymbol{x}_1^{(k)})} \tag{18.62}$$

If the bridge distribution p_* is chosen carefully to have a large overlap of support with both p_0 and p_1, then bridge sampling can allow the distance between two distributions (or more formally, $D_{\mathrm{KL}}(p_0 \| p_1)$) to be much larger than with standard importance sampling.

It can be shown that the optimal bridging distribution is given by $p_*^{(opt)}(\mathbf{x}) \propto \frac{\tilde{p}_0(\boldsymbol{x})\tilde{p}_1(\boldsymbol{x})}{r\tilde{p}_0(\boldsymbol{x})+\tilde{p}_1(\boldsymbol{x})}$ where $r = Z_1/Z_0$. At first, this appears to be an unworkable solution as it would seem to require the very quantity we are trying to estimate, Z_1/Z_0. However, it is possible to start with a coarse estimate of r and use the resulting bridge distribution to refine our estimate iteratively (Neal, 2005). That is, we iteratively re-estimate the ratio and use each iteration to update the value of r.

Linked importance sampling Both AIS and bridge sampling have their advantages. If $D_{\mathrm{KL}}(p_0 \| p_1)$ is not too large (because p_0 and p_1 are sufficiently close) bridge sampling can be a more effective means of estimating the ratio of partition functions than AIS. If, however, the two distributions are too far apart for a single distribution p_* to bridge the gap then one can at least use AIS with potentially many intermediate distributions to span the distance between p_0 and p_1. Neal (2005) showed how his linked importance sampling method leveraged the power of the bridge sampling strategy to bridge the intermediate distributions used in AIS to significantly improve the overall partition function estimates.

Estimating the partition function while training While AIS has become accepted as the standard method for estimating the partition function for many undirected models, it is sufficiently computationally intensive that it remains infeasible to use during training. However, alternative strategies that have been explored to maintain an estimate of the partition function throughout training

Using a combination of bridge sampling, short-chain AIS and parallel tempering, Desjardins *et al.* (2011) devised a scheme to track the partition function of an

RBM throughout the training process. The strategy is based on the maintenance of independent estimates of the partition functions of the RBM at every temperature operating in the parallel tempering scheme. The authors combined bridge sampling estimates of the ratios of partition functions of neighboring chains (i.e. from parallel tempering) with AIS estimates across time to come up with a low variance estimate of the partition functions at every iteration of learning.

The tools described in this chapter provide many different ways of overcoming the problem of intractable partition functions, but there can be several other difficulties involved in training and using generative models. Foremost among these is the problem of intractable inference, which we confront next.

Chapter 19

Approximate Inference

Many probabilistic models are difficult to train because it is difficult to perform inference in them. In the context of deep learning, we usually have a set of visible variables $\boldsymbol{v}$ and a set of latent variables $\boldsymbol{h}$. The challenge of inference usually refers to the difficult problem of computing $p(\boldsymbol{h} \mid \boldsymbol{v})$ or taking expectations with respect to it. Such operations are often necessary for tasks like maximum likelihood learning.

Many simple graphical models with only one hidden layer, such as restricted Boltzmann machines and probabilistic PCA, are defined in a way that makes inference operations like computing $p(\boldsymbol{h} \mid \boldsymbol{v})$, or taking expectations with respect to it, simple. Unfortunately, most graphical models with multiple layers of hidden variables have intractable posterior distributions. Exact inference requires an exponential amount of time in these models. Even some models with only a single layer, such as sparse coding, have this problem.

In this chapter, we introduce several of the techniques for confronting these intractable inference problems. Later, in chapter 20, we will describe how to use these techniques to train probabilistic models that would otherwise be intractable, such as deep belief networks and deep Boltzmann machines.

Intractable inference problems in deep learning usually arise from interactions between latent variables in a structured graphical model. See figure 19.1 for some examples. These interactions may be due to direct interactions in undirected models or "explaining away" interactions between mutual ancestors of the same visible unit in directed models.

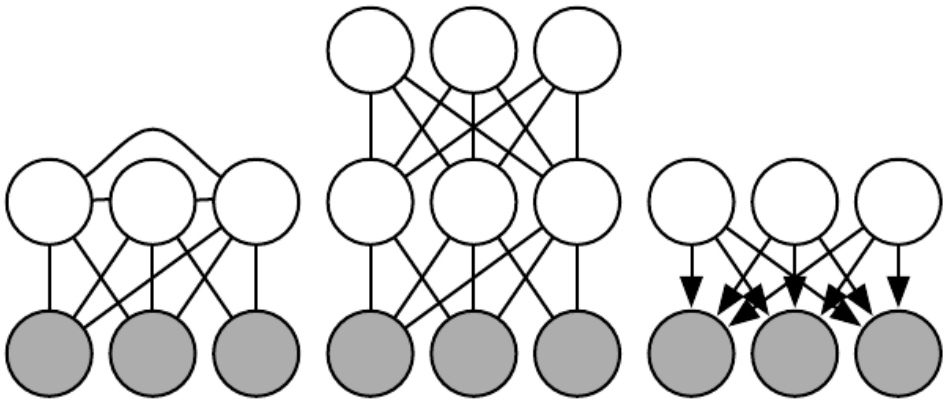

Figure 19.1: Intractable inference problems in deep learning are usually the result of interactions between latent variables in a structured graphical model. These can be due to edges directly connecting one latent variable to another, or due to longer paths that are activated when the child of a V-structure is observed. *(Left)*A **semi-restricted Boltzmann machine** (Osindero and Hinton, 2008) with connections between hidden units. These direct connections between latent variables make the posterior distribution intractable due to large cliques of latent variables. *(Center)*A deep Boltzmann machine, organized into layers of variables without intra-layer connections, still has an intractable posterior distribution due to the connections between layers. *(Right)*This directed model has interactions between latent variables when the visible variables are observed, because every two latent variables are co-parents. Some probabilistic models are able to provide tractable inference over the latent variables despite having one of the graph structures depicted above. This is possible if the conditional probability distributions are chosen to introduce additional independences beyond those described by the graph. For example, probabilistic PCA has the graph structure shown in the right, yet still has simple inference due to special properties of the specific conditional distributions it uses (linear-Gaussian conditionals with mutually orthogonal basis vectors).

19.1 Inference as Optimization

Many approaches to confronting the problem of difficult inference make use of the observation that exact inference can be described as an optimization problem. Approximate inference algorithms may then be derived by approximating the underlying optimization problem.

To construct the optimization problem, assume we have a probabilistic model consisting of observed variables $\boldsymbol{v}$ and latent variables $\boldsymbol{h}$. We would like to compute the log probability of the observed data, $\log p(\boldsymbol{v}; \boldsymbol{\theta})$. Sometimes it is too difficult to compute $\log p(\boldsymbol{v}; \boldsymbol{\theta})$ if it is costly to marginalize out $\boldsymbol{h}$. Instead, we can compute a lower bound $\mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}, q)$ on $\log p(\boldsymbol{v}; \boldsymbol{\theta})$. This bound is called the **evidence lower bound** (ELBO). Another commonly used name for this lower bound is the negative **variational free energy**. Specifically, the evidence lower bound is defined to be

$$\mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}, q) = \log p(\boldsymbol{v}; \boldsymbol{\theta}) - D_{\mathrm{KL}} \left(q(\boldsymbol{h} \mid \boldsymbol{v}) \| p(\boldsymbol{h} \mid \boldsymbol{v}; \boldsymbol{\theta}) \right) \tag{19.1}$$

where q is an arbitrary probability distribution over $\boldsymbol{h}$.

Because the difference between $\log p(\boldsymbol{v})$ and $\mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}, q)$ is given by the KL divergence and because the KL divergence is always non-negative, we can see that $\mathcal{L}$ always has at most the same value as the desired log probability. The two are equal if and only if q is the same distribution as $p(\boldsymbol{h} \mid \boldsymbol{v})$.

Surprisingly, $\mathcal{L}$ can be considerably easier to compute for some distributions q. Simple algebra shows that we can rearrange $\mathcal{L}$ into a much more convenient form:

$$\begin{aligned}
\mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}, q) &= \log p(\boldsymbol{v}; \boldsymbol{\theta}) - D_{\mathrm{KL}}(q(\boldsymbol{h} \mid \boldsymbol{v}) \| p(\boldsymbol{h} \mid \boldsymbol{v}; \boldsymbol{\theta})) && (19.2)\\
&= \log p(\boldsymbol{v}; \boldsymbol{\theta}) - \mathbb{E}_{\mathbf{h} \sim q} \log \frac{q(\boldsymbol{h} \mid \boldsymbol{v})}{p(\boldsymbol{h} \mid \boldsymbol{v})} && (19.3)\\
&= \log p(\boldsymbol{v}; \boldsymbol{\theta}) - \mathbb{E}_{\mathbf{h} \sim q} \log \frac{q(\boldsymbol{h} \mid \boldsymbol{v})}{\frac{p(\boldsymbol{h}, \boldsymbol{v}; \boldsymbol{\theta})}{p(\boldsymbol{v}; \boldsymbol{\theta})}} && (19.4)\\
&= \log p(\boldsymbol{v}; \boldsymbol{\theta}) - \mathbb{E}_{\mathbf{h} \sim q} \left[\log q(\boldsymbol{h} \mid \boldsymbol{v}) - \log p(\boldsymbol{h}, \boldsymbol{v}; \boldsymbol{\theta}) + \log p(\boldsymbol{v}; \boldsymbol{\theta}) \right] && (19.5)\\
&= -\mathbb{E}_{\mathbf{h} \sim q} \left[\log q(\boldsymbol{h} \mid \boldsymbol{v}) - \log p(\boldsymbol{h}, \boldsymbol{v}; \boldsymbol{\theta}) \right]. && (19.6)
\end{aligned}$$

This yields the more canonical definition of the evidence lower bound,

$$\mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}, q) = \mathbb{E}_{\mathbf{h} \sim q} \left[\log p(\boldsymbol{h}, \boldsymbol{v}) \right] + H(q). \tag{19.7}$$

For an appropriate choice of q, $\mathcal{L}$ is tractable to compute. For any choice of q, $\mathcal{L}$ provides a lower bound on the likelihood. For $q(\boldsymbol{h} \mid \boldsymbol{v})$ that are better

approximations of $p(\boldsymbol{h} \mid \boldsymbol{v})$, the lower bound $\mathcal{L}$ will be tighter, in other words, closer to $\log p(\boldsymbol{v})$. When $q(\boldsymbol{h} \mid \boldsymbol{v}) = p(\boldsymbol{h} \mid \boldsymbol{v})$, the approximation is perfect, and $\mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}, q) = \log p(\boldsymbol{v}; \boldsymbol{\theta})$.

We can thus think of inference as the procedure for finding the q that maximizes $\mathcal{L}$. Exact inference maximizes $\mathcal{L}$ perfectly by searching over a family of functions q that includes $p(\boldsymbol{h} \mid \boldsymbol{v})$. Throughout this chapter, we will show how to derive different forms of approximate inference by using approximate optimization to find q. We can make the optimization procedure less expensive but approximate by restricting the family of distributions q the optimization is allowed to search over or by using an imperfect optimization procedure that may not completely maximize $\mathcal{L}$ but merely increase it by a significant amount.

No matter what choice of q we use, $\mathcal{L}$ is a lower bound. We can get tighter or looser bounds that are cheaper or more expensive to compute depending on how we choose to approach this optimization problem. We can obtain a poorly matched q but reduce the computational cost by using an imperfect optimization procedure, or by using a perfect optimization procedure over a restricted family of q distributions.

19.2 Expectation Maximization

The first algorithm we introduce based on maximizing a lower bound $\mathcal{L}$ is the **expectation maximization** (EM) algorithm, a popular training algorithm for models with latent variables. We describe here a view on the EM algorithm developed by Neal and Hinton (1999). Unlike most of the other algorithms we describe in this chapter, EM is not an approach to approximate inference, but rather an approach to learning with an approximate posterior.

The EM algorithm consists of alternating between two steps until convergence:

- The **E-step** (Expectation step): Let $\boldsymbol{\theta}^{(0)}$ denote the value of the parameters at the beginning of the step. Set $q(\boldsymbol{h}^{(i)} \mid \boldsymbol{v}) = p(\boldsymbol{h}^{(i)} \mid \boldsymbol{v}^{(i)}; \boldsymbol{\theta}^{(0)})$ for all indices i of the training examples $\boldsymbol{v}^{(i)}$ we want to train on (both batch and minibatch variants are valid). By this we mean q is defined in terms of the *current* parameter value of $\boldsymbol{\theta}^{(0)}$; if we vary $\boldsymbol{\theta}$ then $p(\boldsymbol{h} \mid \boldsymbol{v}; \boldsymbol{\theta})$ will change but $q(\boldsymbol{h} \mid \boldsymbol{v})$ will remain equal to $p(\boldsymbol{h} \mid \boldsymbol{v}; \boldsymbol{\theta}^{(0)})$.
- The **M-step** (Maximization step): Completely or partially maximize

$$\sum_i \mathcal{L}(\boldsymbol{v}^{(i)}, \boldsymbol{\theta}, q) \tag{19.8}$$

with respect to $\boldsymbol{\theta}$ using your optimization algorithm of choice.

This can be viewed as a coordinate ascent algorithm to maximize $\mathcal{L}$. On one step, we maximize $\mathcal{L}$ with respect to q, and on the other, we maximize $\mathcal{L}$ with respect to $\boldsymbol{\theta}$.

Stochastic gradient ascent on latent variable models can be seen as a special case of the EM algorithm where the M step consists of taking a single gradient step. Other variants of the EM algorithm can make much larger steps. For some model families, the M step can even be performed analytically, jumping all the way to the optimal solution for $\boldsymbol{\theta}$ given the current q.

Even though the E-step involves exact inference, we can think of the EM algorithm as using approximate inference in some sense. Specifically, the M-step assumes that the same value of q can be used for all values of $\boldsymbol{\theta}$. This will introduce a gap between $\mathcal{L}$ and the true $\log p(\boldsymbol{v})$ as the M-step moves further and further away from the value $\boldsymbol{\theta}^{(0)}$ used in the E-step. Fortunately, the E-step reduces the gap to zero again as we enter the loop for the next time.

The EM algorithm contains a few different insights. First, there is the basic structure of the learning process, in which we update the model parameters to improve the likelihood of a completed dataset, where all missing variables have their values provided by an estimate of the posterior distribution. This particular insight is not unique to the EM algorithm. For example, using gradient descent to maximize the log-likelihood also has this same property; the log-likelihood gradient computations require taking expectations with respect to the posterior distribution over the hidden units. Another key insight in the EM algorithm is that we can continue to use one value of q even after we have moved to a different value of $\boldsymbol{\theta}$. This particular insight is used throughout classical machine learning to derive large M-step updates. In the context of deep learning, most models are too complex to admit a tractable solution for an optimal large M-step update, so this second insight which is more unique to the EM algorithm is rarely used.

19.3 MAP Inference and Sparse Coding

We usually use the term inference to refer to computing the probability distribution over one set of variables given another. When training probabilistic models with latent variables, we are usually interested in computing $p(\boldsymbol{h} \mid \boldsymbol{v})$. An alternative form of inference is to compute the single most likely value of the missing variables, rather than to infer the entire distribution over their possible values. In the context

of latent variable models, this means computing

$$\boldsymbol{h}^* = \arg\max_{\boldsymbol{h}} p(\boldsymbol{h} \mid \boldsymbol{v}). \tag{19.9}$$

This is known as **maximum a posteriori** inference, abbreviated MAP inference.

MAP inference is usually not thought of as approximate inference—it does compute the exact most likely value of $\boldsymbol{h}^*$. However, if we wish to develop a learning process based on maximizing $\mathcal{L}(\boldsymbol{v}, \boldsymbol{h}, q)$, then it is helpful to think of MAP inference as a procedure that provides a value of q. In this sense, we can think of MAP inference as approximate inference, because it does not provide the optimal q.

Recall from section 19.1 that exact inference consists of maximizing

$$\mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}, q) = \mathbb{E}_{\mathbf{h} \sim q} \left[\log p(\boldsymbol{h}, \boldsymbol{v}) \right] + H(q) \tag{19.10}$$

with respect to q over an unrestricted family of probability distributions, using an exact optimization algorithm. We can derive MAP inference as a form of approximate inference by restricting the family of distributions q may be drawn from. Specifically, we require q to take on a Dirac distribution:

$$q(\boldsymbol{h} \mid \boldsymbol{v}) = \delta(\boldsymbol{h} - \boldsymbol{\mu}). \tag{19.11}$$

This means that we can now control q entirely via $\boldsymbol{\mu}$. Dropping terms of $\mathcal{L}$ that do not vary with $\boldsymbol{\mu}$, we are left with the optimization problem

$$\boldsymbol{\mu}^* = \arg\max_{\boldsymbol{\mu}} \log p(\boldsymbol{h} = \boldsymbol{\mu}, \boldsymbol{v}), \tag{19.12}$$

which is equivalent to the MAP inference problem

$$\boldsymbol{h}^* = \arg\max_{\boldsymbol{h}} p(\boldsymbol{h} \mid \boldsymbol{v}). \tag{19.13}$$

We can thus justify a learning procedure similar to EM, in which we alternate between performing MAP inference to infer $\boldsymbol{h}^*$ and then update $\boldsymbol{\theta}$ to increase $\log p(\boldsymbol{h}^*, \boldsymbol{v})$. As with EM, this is a form of coordinate ascent on $\mathcal{L}$, where we alternate between using inference to optimize $\mathcal{L}$ with respect to q and using parameter updates to optimize $\mathcal{L}$ with respect to $\boldsymbol{\theta}$. The procedure as a whole can be justified by the fact that $\mathcal{L}$ is a lower bound on $\log p(\boldsymbol{v})$. In the case of MAP inference, this justification is rather vacuous, because the bound is infinitely loose, due to the Dirac distribution's differential entropy of negative infinity. However, adding noise to $\boldsymbol{\mu}$ would make the bound meaningful again.

MAP inference is commonly used in deep learning as both a feature extractor and a learning mechanism. It is primarily used for sparse coding models.

Recall from section 13.4 that sparse coding is a linear factor model that imposes a sparsity-inducing prior on its hidden units. A common choice is a factorial Laplace prior, with

$$p(h_i) = \frac{\lambda}{2} e^{-\lambda |h_i|}. \tag{19.14}$$

The visible units are then generated by performing a linear transformation and adding noise:

$$p(\boldsymbol{x} \mid \boldsymbol{h}) = \mathcal{N}(\boldsymbol{v}; \boldsymbol{W}\boldsymbol{h} + \boldsymbol{b}, \beta^{-1}\boldsymbol{I}). \tag{19.15}$$

Computing or even representing $p(\boldsymbol{h} \mid \boldsymbol{v})$ is difficult. Every pair of variables h_i and h_j are both parents of $\boldsymbol{v}$. This means that when $\boldsymbol{v}$ is observed, the graphical model contains an active path connecting h_i and h_j. All of the hidden units thus participate in one massive clique in $p(\boldsymbol{h} \mid \boldsymbol{v})$. If the model were Gaussian then these interactions could be modeled efficiently via the covariance matrix, but the sparse prior makes these interactions non-Gaussian.

Because $p(\boldsymbol{h} \mid \boldsymbol{v})$ is intractable, so is the computation of the log-likelihood and its gradient. We thus cannot use exact maximum likelihood learning. Instead, we use MAP inference and learn the parameters by maximizing the ELBO defined by the Dirac distribution around the MAP estimate of $\boldsymbol{h}$.

If we concatenate all of the $\boldsymbol{h}$ vectors in the training set into a matrix $\boldsymbol{H}$, and concatenate all of the $\boldsymbol{v}$ vectors into a matrix $\boldsymbol{V}$, then the sparse coding learning process consists of minimizing

$$J(\boldsymbol{H}, \boldsymbol{W}) = \sum_{i,j} |H_{i,j}| + \sum_{i,j} \left(\boldsymbol{V} - \boldsymbol{H}\boldsymbol{W}^\top \right)^2_{i,j}. \tag{19.16}$$

Most applications of sparse coding also involve weight decay or a constraint on the norms of the columns of $\boldsymbol{W}$, in order to prevent the pathological solution with extremely small $\boldsymbol{H}$ and large $\boldsymbol{W}$.

We can minimize J by alternating between minimization with respect to $\boldsymbol{H}$ and minimization with respect to $\boldsymbol{W}$. Both sub-problems are convex. In fact, the minimization with respect to $\boldsymbol{W}$ is just a linear regression problem. However, minimization of J with respect to both arguments is usually not a convex problem.

Minimization with respect to $\boldsymbol{H}$ requires specialized algorithms such as the feature-sign search algorithm (Lee *et al.*, 2007).

19.4 Variational Inference and Learning

We have seen how the evidence lower bound $\mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}, q)$ is a lower bound on $\log p(\boldsymbol{v}; \boldsymbol{\theta})$, how inference can be viewed as maximizing $\mathcal{L}$ with respect to q, and how learning can be viewed as maximizing $\mathcal{L}$ with respect to $\boldsymbol{\theta}$. We have seen that the EM algorithm allows us to make large learning steps with a fixed q and that learning algorithms based on MAP inference allow us to learn using a point estimate of $p(\boldsymbol{h} \mid \boldsymbol{v})$ rather than inferring the entire distribution. Now we develop the more general approach to variational learning.

The core idea behind variational learning is that we can maximize $\mathcal{L}$ over a restricted family of distributions q. This family should be chosen so that it is easy to compute $\mathbb{E}_q \log p(\boldsymbol{h}, \boldsymbol{v})$. A typical way to do this is to introduce assumptions about how q factorizes.

A common approach to variational learning is to impose the restriction that q is a factorial distribution:

$$q(\boldsymbol{h} \mid \boldsymbol{v}) = \prod_i q(h_i \mid \boldsymbol{v}). \tag{19.17}$$

This is called the **mean field** approach. More generally, we can impose any graphical model structure we choose on q, to flexibly determine how many interactions we want our approximation to capture. This fully general graphical model approach is called **structured variational inference** (Saul and Jordan, 1996).

The beauty of the variational approach is that we do not need to specify a specific parametric form for q. We specify how it should factorize, but then the optimization problem determines the optimal probability distribution within those factorization constraints. For discrete latent variables, this just means that we use traditional optimization techniques to optimize a finite number of variables describing the q distribution. For continuous latent variables, this means that we use a branch of mathematics called calculus of variations to perform optimization over a space of functions, and actually determine which function should be used to represent q. Calculus of variations is the origin of the names "variational learning" and "variational inference," though these names apply even when the latent variables are discrete and calculus of variations is not needed. In the case of continuous latent variables, calculus of variations is a powerful technique that removes much of the responsibility from the human designer of the model, who now must specify only how q factorizes, rather than needing to guess how to design a specific q that can accurately approximate the posterior.

Because $\mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}, q)$ is defined to be $\log p(\boldsymbol{v}; \boldsymbol{\theta}) - D_{\mathrm{KL}}(q(\boldsymbol{h} \mid \boldsymbol{v}) \| p(\boldsymbol{h} \mid \boldsymbol{v}; \boldsymbol{\theta}))$, we can think of maximizing $\mathcal{L}$ with respect to q as minimizing $D_{\mathrm{KL}}(q(\boldsymbol{h} \mid \boldsymbol{v}) \| p(\boldsymbol{h} \mid \boldsymbol{v}))$.

In this sense, we are fitting q to p. However, we are doing so with the opposite direction of the KL divergence than we are used to using for fitting an approximation. When we use maximum likelihood learning to fit a model to data, we minimize $D_{\mathrm{KL}}(p_{\mathrm{data}} \| p_{\mathrm{model}})$. As illustrated in figure 3.6, this means that maximum likelihood encourages the model to have high probability everywhere that the data has high probability, while our optimization-based inference procedure encourages q to have low probability everywhere the true posterior has low probability. Both directions of the KL divergence can have desirable and undesirable properties. The choice of which to use depends on which properties are the highest priority for each application. In the case of the inference optimization problem, we choose to use $D_{\mathrm{KL}}(q(\boldsymbol{h} \mid \boldsymbol{v}) \| p(\boldsymbol{h} \mid \boldsymbol{v}))$ for computational reasons. Specifically, computing $D_{\mathrm{KL}}(q(\boldsymbol{h} \mid \boldsymbol{v}) \| p(\boldsymbol{h} \mid \boldsymbol{v}))$ involves evaluating expectations with respect to q, so by designing q to be simple, we can simplify the required expectations. The opposite direction of the KL divergence would require computing expectations with respect to the true posterior. Because the form of the true posterior is determined by the choice of model, we cannot design a reduced-cost approach to computing $D_{\mathrm{KL}}(p(\boldsymbol{h} \mid \boldsymbol{v}) \| q(\boldsymbol{h} \mid \boldsymbol{v}))$ exactly.

19.4.1 Discrete Latent Variables

Variational inference with discrete latent variables is relatively straightforward. We define a distribution q, typically one where each factor of q is just defined by a lookup table over discrete states. In the simplest case, $\boldsymbol{h}$ is binary and we make the mean field assumption that q factorizes over each individual h_i. In this case we can parametrize q with a vector $\hat{\boldsymbol{h}}$ whose entries are probabilities. Then $q(h_i = 1 \mid \boldsymbol{v}) = \hat{h}_i$.

After determining how to represent q, we simply optimize its parameters. In the case of discrete latent variables, this is just a standard optimization problem. In principle the selection of q could be done with any optimization algorithm, such as gradient descent.

Because this optimization must occur in the inner loop of a learning algorithm, it must be very fast. To achieve this speed, we typically use special optimization algorithms that are designed to solve comparatively small and simple problems in very few iterations. A popular choice is to iterate fixed point equations, in other words, to solve

$$\frac{\partial}{\partial \hat{h}_i} \mathcal{L} = 0 \tag{19.18}$$

for $\hat{h}_i$. We repeatedly update different elements of $\hat{\boldsymbol{h}}$ until we satisfy a convergence

criterion.

To make this more concrete, we show how to apply variational inference to the **binary sparse coding** model (we present here the model developed by Henniges *et al.* (2010) but demonstrate traditional, generic mean field applied to the model, while they introduce a specialized algorithm). This derivation goes into considerable mathematical detail and is intended for the reader who wishes to fully resolve any ambiguity in the high-level conceptual description of variational inference and learning we have presented so far. Readers who do not plan to derive or implement variational learning algorithms may safely skip to the next section without missing any new high-level concepts. Readers who proceed with the binary sparse coding example are encouraged to review the list of useful properties of functions that commonly arise in probabilistic models in section 3.10. We use these properties liberally throughout the following derivations without highlighting exactly where we use each one.

In the binary sparse coding model, the input $\boldsymbol{v} \in \mathbb{R}^n$ is generated from the model by adding Gaussian noise to the sum of m different components which can each be present or absent. Each component is switched on or off by the corresponding hidden unit in $\boldsymbol{h} \in \{0, 1\}^m$:

$$p(h_i = 1) = \sigma(b_i) \tag{19.19}$$

$$p(\boldsymbol{v} \mid \boldsymbol{h}) = \mathcal{N}(\boldsymbol{v}; \boldsymbol{W}\boldsymbol{h}, \boldsymbol{\beta}^{-1}) \tag{19.20}$$

where $\boldsymbol{b}$ is a learnable set of biases, $\boldsymbol{W}$ is a learnable weight matrix, and $\boldsymbol{\beta}$ is a learnable, diagonal precision matrix.

Training this model with maximum likelihood requires taking the derivative with respect to the parameters. Consider the derivative with respect to one of the biases:

$$\frac{\partial}{\partial b_i} \log p(\boldsymbol{v}) \tag{19.21}$$

$$= \frac{\frac{\partial}{\partial b_i} p(\boldsymbol{v})}{p(\boldsymbol{v})} \tag{19.22}$$

$$= \frac{\frac{\partial}{\partial b_i} \sum_{\boldsymbol{h}} p(\boldsymbol{h}, \boldsymbol{v})}{p(\boldsymbol{v})} \tag{19.23}$$

$$= \frac{\frac{\partial}{\partial b_i} \sum_{\boldsymbol{h}} p(\boldsymbol{h}) p(\boldsymbol{v} \mid \boldsymbol{h})}{p(\boldsymbol{v})} \tag{19.24}$$

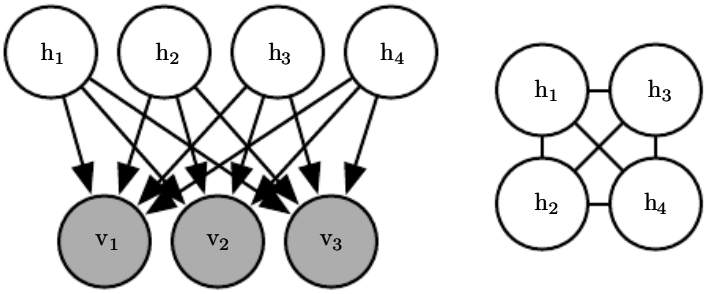

Figure 19.2: The graph structure of a binary sparse coding model with four hidden units. *(Left)*The graph structure of $p(\boldsymbol{h}, \boldsymbol{v})$. Note that the edges are directed, and that every two hidden units are co-parents of every visible unit. *(Right)*The graph structure of $p(\boldsymbol{h} \mid \boldsymbol{v})$. In order to account for the active paths between co-parents, the posterior distribution needs an edge between all of the hidden units.

$$= \frac{\sum_{\boldsymbol{h}} p(\boldsymbol{v} \mid \boldsymbol{h}) \frac{\partial}{\partial b_i} p(\boldsymbol{h})}{p(\boldsymbol{v})} \tag{19.25}$$

$$= \sum_{\boldsymbol{h}} p(\boldsymbol{h} \mid \boldsymbol{v}) \frac{\frac{\partial}{\partial b_i} p(\boldsymbol{h})}{p(\boldsymbol{h})} \tag{19.26}$$

$$= \mathbb{E}_{\mathbf{h} \sim p(\boldsymbol{h} \mid \boldsymbol{v})} \frac{\partial}{\partial b_i} \log p(\boldsymbol{h}). \tag{19.27}$$

This requires computing expectations with respect to $p(\boldsymbol{h} \mid \boldsymbol{v})$. Unfortunately, $p(\boldsymbol{h} \mid \boldsymbol{v})$ is a complicated distribution. See figure 19.2 for the graph structure of $p(\boldsymbol{h}, \boldsymbol{v})$ and $p(\boldsymbol{h} \mid \boldsymbol{v})$. The posterior distribution corresponds to the complete graph over the hidden units, so variable elimination algorithms do not help us to compute the required expectations any faster than brute force.

We can resolve this difficulty by using variational inference and variational learning instead.

We can make a mean field approximation:

$$q(\boldsymbol{h} \mid \boldsymbol{v}) = \prod_i q(h_i \mid \boldsymbol{v}). \tag{19.28}$$

The latent variables of the binary sparse coding model are binary, so to represent a factorial q we simply need to model m Bernoulli distributions $q(h_i \mid \boldsymbol{v})$. A natural way to represent the means of the Bernoulli distributions is with a vector $\hat{\boldsymbol{h}}$ of probabilities, with $q(h_i = 1 \mid \boldsymbol{v}) = \hat{h}_i$. We impose a restriction that $\hat{h}_i$ is never equal to 0 or to 1, in order to avoid errors when computing, for example, $\log \hat{h}_i$.

We will see that the variational inference equations never assign 0 or 1 to $\hat{h}_i$

analytically. However, in a software implementation, machine rounding error could result in 0 or 1 values. In software, we may wish to implement binary sparse coding using an unrestricted vector of variational parameters $\boldsymbol{z}$ and obtain $\hat{\boldsymbol{h}}$ via the relation $\hat{\boldsymbol{h}} = \sigma(\boldsymbol{z})$. We can thus safely compute $\log \hat{\boldsymbol{h}}_i$ on a computer by using the identity $\log \sigma(z_i) = -\zeta(-z_i)$ relating the sigmoid and the softplus.

To begin our derivation of variational learning in the binary sparse coding model, we show that the use of this mean field approximation makes learning tractable.

The evidence lower bound is given by

$$\mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}, q) \tag{19.29}$$

$$= \mathbb{E}_{\mathbf{h}\sim q}[\log p(\boldsymbol{h}, \boldsymbol{v})] + H(q) \tag{19.30}$$

$$= \mathbb{E}_{\mathbf{h}\sim q}[\log p(\boldsymbol{h}) + \log p(\boldsymbol{v} \mid \boldsymbol{h}) - \log q(\boldsymbol{h} \mid \boldsymbol{v})] \tag{19.31}$$

$$= \mathbb{E}_{\mathbf{h}\sim q}\left[\sum_{i=1}^{m} \log p(h_i) + \sum_{i=1}^{n} \log p(v_i \mid \boldsymbol{h}) - \sum_{i=1}^{m} \log q(h_i \mid \boldsymbol{v})\right] \tag{19.32}$$

$$= \sum_{i=1}^{m} \left[\hat{h}_i(\log \sigma(b_i) - \log \hat{h}_i) + (1 - \hat{h}_i)(\log \sigma(-b_i) - \log(1 - \hat{h}_i))\right] \tag{19.33}$$

$$+ \mathbb{E}_{\mathbf{h}\sim q}\left[\sum_{i=1}^{n} \log \sqrt{\frac{\beta_i}{2\pi}} \exp\left(-\frac{\beta_i}{2}(v_i - \boldsymbol{W}_{i,:}\boldsymbol{h})^2\right)\right] \tag{19.34}$$

$$= \sum_{i=1}^{m} \left[\hat{h}_i(\log \sigma(b_i) - \log \hat{h}_i) + (1 - \hat{h}_i)(\log \sigma(-b_i) - \log(1 - \hat{h}_i))\right] \tag{19.35}$$

$$+ \frac{1}{2}\sum_{i=1}^{n}\left[\log \frac{\beta_i}{2\pi} - \beta_i\left(v_i^2 - 2v_i \boldsymbol{W}_{i,:}\hat{\boldsymbol{h}} + \sum_j \left[W_{i,j}^2 \hat{h_j} + \sum_{k\neq j} W_{i,j}W_{i,k}\hat{h}_j\hat{h}_k\right]\right)\right]. \tag{19.36}$$

While these equations are somewhat unappealing aesthetically, they show that $\mathcal{L}$ can be expressed in a small number of simple arithmetic operations. The evidence lower bound $\mathcal{L}$ is therefore tractable. We can use $\mathcal{L}$ as a replacement for the intractable log-likelihood.

In principle, we could simply run gradient ascent on both $\boldsymbol{v}$ and $\boldsymbol{h}$ and this would make a perfectly acceptable combined inference and training algorithm. Usually, however, we do not do this, for two reasons. First, this would require storing $\hat{\boldsymbol{h}}$ for each $\boldsymbol{v}$. We typically prefer algorithms that do not require per-example memory. It is difficult to scale learning algorithms to billions of examples if we must remember a dynamically updated vector associated with each example.

Second, we would like to be able to extract the features $\hat{\boldsymbol{h}}$ very quickly, in order to recognize the content of $\boldsymbol{v}$. In a realistic deployed setting, we would need to be able to compute $\hat{\boldsymbol{h}}$ in real time.

For both these reasons, we typically do not use gradient descent to compute the mean field parameters $\hat{\boldsymbol{h}}$. Instead, we rapidly estimate them with fixed point equations.

The idea behind fixed point equations is that we are seeking a local maximum with respect to $\hat{\boldsymbol{h}}$, where $\nabla_{\boldsymbol{h}} \mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}, \hat{\boldsymbol{h}}) = \boldsymbol{0}$. We cannot efficiently solve this equation with respect to all of $\hat{\boldsymbol{h}}$ simultaneously. However, we can solve for a single variable:

$$\frac{\partial}{\partial \hat{h}_i} \mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}, \hat{\boldsymbol{h}}) = 0. \tag{19.37}$$

We can then iteratively apply the solution to the equation for $i = 1, \ldots, m$, and repeat the cycle until we satisfy a converge criterion. Common convergence criteria include stopping when a full cycle of updates does not improve $\mathcal{L}$ by more than some tolerance amount, or when the cycle does not change $\hat{\boldsymbol{h}}$ by more than some amount.

Iterating mean field fixed point equations is a general technique that can provide fast variational inference in a broad variety of models. To make this more concrete, we show how to derive the updates for the binary sparse coding model in particular.

First, we must write an expression for the derivatives with respect to $\hat{h}_i$. To do so, we substitute equation 19.36 into the left side of equation 19.37:

$$\frac{\partial}{\partial \hat{h}_i} \mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}, \hat{\boldsymbol{h}}) \tag{19.38}$$

$$= \frac{\partial}{\partial \hat{h}_i} \left[\sum_{j=1}^{m} \left[\hat{h_j} (\log \sigma(b_j) - \log \hat{h_j}) + (1 - \hat{h_j})(\log \sigma(-b_j) - \log(1 - \hat{h_j})) \right] \right. \tag{19.39}$$

$$\left. + \frac{1}{2} \sum_{j=1}^{n} \left[\log \frac{\beta_j}{2\pi} - \beta_j \left(v_j^2 - 2 v_j \boldsymbol{W}_{j,:} \hat{\boldsymbol{h}} + \sum_k \left[W_{j,k}^2 \hat{h}_k + \sum_{l \neq k} W_{j,k} W_{j,l} \hat{h}_k \hat{h}_l \right] \right) \right] \right] \tag{19.40}$$

$$= \log \sigma(b_i) - \log \hat{h}_i - 1 + \log(1 - \hat{h}_i) + 1 - \log \sigma(-b_i) \tag{19.41}$$

$$+ \sum_{j=1}^{n} \left[\beta_j \left(v_j W_{j,i} - \frac{1}{2} W_{j,i}^2 - \sum_{k \neq i} \boldsymbol{W}_{j,k} \boldsymbol{W}_{j,i} \hat{h}_k \right) \right] \tag{19.42}$$

$$= b_i - \log \hat{h}_i + \log(1 - \hat{h}_i) + \boldsymbol{v}^\top \boldsymbol{\beta} \boldsymbol{W}_{:,i} - \frac{1}{2} \boldsymbol{W}_{:,i}^\top \boldsymbol{\beta} \boldsymbol{W}_{:,i} - \sum_{j \neq i} \boldsymbol{W}_{:,j}^\top \boldsymbol{\beta} \boldsymbol{W}_{:,i} \hat{h}_j. \quad (19.43)$$

To apply the fixed point update inference rule, we solve for the $\hat{h}_i$ that sets equation 19.43 to 0:

$$\hat{h}_i = \sigma \left(b_i + \boldsymbol{v}^\top \boldsymbol{\beta} \boldsymbol{W}_{:,i} - \frac{1}{2} \boldsymbol{W}_{:,i}^\top \boldsymbol{\beta} \boldsymbol{W}_{:,i} - \sum_{j \neq i} \boldsymbol{W}_{:,j}^\top \boldsymbol{\beta} \boldsymbol{W}_{:,i} \hat{h}_j \right). \quad (19.44)$$

At this point, we can see that there is a close connection between recurrent neural networks and inference in graphical models. Specifically, the mean field fixed point equations defined a recurrent neural network. The task of this network is to perform inference. We have described how to derive this network from a model description, but it is also possible to train the inference network directly. Several ideas based on this theme are described in chapter 20.

In the case of binary sparse coding, we can see that the recurrent network connection specified by equation 19.44 consists of repeatedly updating the hidden units based on the changing values of the neighboring hidden units. The input always sends a fixed message of $\boldsymbol{v}^\top \beta \boldsymbol{W}$ to the hidden units, but the hidden units constantly update the message they send to each other. Specifically, two units $\hat{h}_i$ and $\hat{h}_j$ inhibit each other when their weight vectors are aligned. This is a form of competition—between two hidden units that both explain the input, only the one that explains the input best will be allowed to remain active. This competition is the mean field approximation's attempt to capture the explaining away interactions in the binary sparse coding posterior. The explaining away effect actually should cause a multi-modal posterior, so that if we draw samples from the posterior, some samples will have one unit active, other samples will have the other unit active, but very few samples have both active. Unfortunately, explaining away interactions cannot be modeled by the factorial q used for mean field, so the mean field approximation is forced to choose one mode to model. This is an instance of the behavior illustrated in figure 3.6.

We can rewrite equation 19.44 into an equivalent form that reveals some further insights:

$$\hat{h}_i = \sigma \left(b_i + \left(\boldsymbol{v} - \sum_{j \neq i} \boldsymbol{W}_{:,j} \hat{h}_j \right)^\top \boldsymbol{\beta} \boldsymbol{W}_{:,i} - \frac{1}{2} \boldsymbol{W}_{:,i}^\top \boldsymbol{\beta} \boldsymbol{W}_{:,i} \right). \quad (19.45)$$

In this reformulation, we see the input at each step as consisting of $\boldsymbol{v} - \sum_{j \neq i} \boldsymbol{W}_{:,j} \hat{h}_j$ rather than $\boldsymbol{v}$. We can thus think of unit i as attempting to encode the residual

error in $\boldsymbol{v}$ given the code of the other units. We can thus think of sparse coding as an iterative autoencoder, that repeatedly encodes and decodes its input, attempting to fix mistakes in the reconstruction after each iteration.

In this example, we have derived an update rule that updates a single unit at a time. It would be advantageous to be able to update more units simultaneously. Some graphical models, such as deep Boltzmann machines, are structured in such a way that we can solve for many entries of $\hat{\boldsymbol{h}}$ simultaneously. Unfortunately, binary sparse coding does not admit such block updates. Instead, we can use a heuristic technique called **damping** to perform block updates. In the damping approach, we solve for the individually optimal values of every element of $\hat{\boldsymbol{h}}$, then move all of the values in a small step in that direction. This approach is no longer guaranteed to increase $\mathcal{L}$ at each step, but works well in practice for many models. See Koller and Friedman (2009) for more information about choosing the degree of synchrony and damping strategies in message passing algorithms.

19.4.2 Calculus of Variations

Before continuing with our presentation of variational learning, we must briefly introduce an important set of mathematical tools used in variational learning: **calculus of variations**.

Many machine learning techniques are based on minimizing a function $J(\boldsymbol{\theta})$ by finding the input vector $\boldsymbol{\theta} \in \mathbb{R}^n$ for which it takes on its minimal value. This can be accomplished with multivariate calculus and linear algebra, by solving for the critical points where $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbf{0}$. In some cases, we actually want to solve for a function $f(\boldsymbol{x})$, such as when we want to find the probability density function over some random variable. This is what calculus of variations enables us to do.

A function of a function f is known as a **functional** $J[f]$. Much as we can take partial derivatives of a function with respect to elements of its vector-valued argument, we can take **functional derivatives**, also known as **variational derivatives**, of a functional $J[f]$ with respect to individual values of the function $f(\boldsymbol{x})$ at any specific value of $\boldsymbol{x}$. The functional derivative of the functional J with respect to the value of the function f at point $\boldsymbol{x}$ is denoted $\frac{\delta}{\delta f(x)} J$.

A complete formal development of functional derivatives is beyond the scope of this book. For our purposes, it is sufficient to state that for differentiable functions $f(\boldsymbol{x})$ and differentiable functions $g(y, \boldsymbol{x})$ with continuous derivatives, that

$$\frac{\delta}{\delta f(\boldsymbol{x})} \int g\left(f(\boldsymbol{x}), \boldsymbol{x}\right) d\boldsymbol{x} = \frac{\partial}{\partial y} g(f(\boldsymbol{x}), \boldsymbol{x}). \tag{19.46}$$

To gain some intuition for this identity, one can think of $f(\boldsymbol{x})$ as being a vector with uncountably many elements, indexed by a real vector $\boldsymbol{x}$. In this (somewhat incomplete view), the identity providing the functional derivatives is the same as we would obtain for a vector $\boldsymbol{\theta} \in \mathbb{R}^n$ indexed by positive integers:

$$\frac{\partial}{\partial \theta_i} \sum_j g(\theta_j, j) = \frac{\partial}{\partial \theta_i} g(\theta_i, i). \tag{19.47}$$

Many results in other machine learning publications are presented using the more general **Euler-Lagrange equation** which allows g to depend on the derivatives of f as well as the value of f, but we do not need this fully general form for the results presented in this book.

To optimize a function with respect to a vector, we take the gradient of the function with respect to the vector and solve for the point where every element of the gradient is equal to zero. Likewise, we can optimize a functional by solving for the function where the functional derivative at every point is equal to zero.

As an example of how this process works, consider the problem of finding the probability distribution function over $x \in \mathbb{R}$ that has maximal differential entropy. Recall that the entropy of a probability distribution $p(x)$ is defined as

$$H[p] = -\mathbb{E}_x \log p(x). \tag{19.48}$$

For continuous values, the expectation is an integral:

$$H[p] = -\int p(x) \log p(x) dx. \tag{19.49}$$

We cannot simply maximize $H[p]$ with respect to the function $p(x)$, because the result might not be a probability distribution. Instead, we need to use Lagrange multipliers to add a constraint that $p(x)$ integrates to 1. Also, the entropy increases without bound as the variance increases. This makes the question of which distribution has the greatest entropy uninteresting. Instead, we ask which distribution has maximal entropy for fixed variance σ^2. Finally, the problem is underdetermined because the distribution can be shifted arbitrarily without changing the entropy. To impose a unique solution, we add a constraint that the mean of the distribution be μ. The Lagrangian functional for this optimization problem is

$$\mathcal{L}[p] = \lambda_1 \left(\int p(x) dx - 1 \right) + \lambda_2 \left(\mathbb{E}[x] - \mu \right) + \lambda_3 \left(\mathbb{E}[(x-\mu)^2] - \sigma^2 \right) + H[p] \tag{19.50}$$

$$= \int \left(\lambda_1 p(x) + \lambda_2 p(x) x + \lambda_3 p(x)(x-\mu)^2 - p(x)\log p(x)\right) dx - \lambda_1 - \mu\lambda_2 - \sigma^2\lambda_3. \tag{19.51}$$

To minimize the Lagrangian with respect to p, we set the functional derivatives equal to 0:

$$\forall x, \frac{\delta}{\delta p(x)}\mathcal{L} = \lambda_1 + \lambda_2 x + \lambda_3(x-\mu)^2 - 1 - \log p(x) = 0. \tag{19.52}$$

This condition now tells us the functional form of $p(x)$. By algebraically re-arranging the equation, we obtain

$$p(x) = \exp\left(\lambda_1 + \lambda_2 x + \lambda_3(x-\mu)^2 - 1\right). \tag{19.53}$$

We never assumed directly that $p(x)$ would take this functional form; we obtained the expression itself by analytically minimizing a functional. To finish the minimization problem, we must choose the λ values to ensure that all of our constraints are satisfied. We are free to choose any λ values, because the gradient of the Lagrangian with respect to the λ variables is zero so long as the constraints are satisfied. To satisfy all of the constraints, we may set $\lambda_1 = 1 - \log\sigma\sqrt{2\pi}$, $\lambda_2 = 0$, and $\lambda_3 = -\frac{1}{2\sigma^2}$ to obtain

$$p(x) = \mathcal{N}(x; \mu, \sigma^2). \tag{19.54}$$

This is one reason for using the normal distribution when we do not know the true distribution. Because the normal distribution has the maximum entropy, we impose the least possible amount of structure by making this assumption.

While examining the critical points of the Lagrangian functional for the entropy, we found only one critical point, corresponding to maximizing the entropy for fixed variance. What about the probability distribution function that *minimizes* the entropy? Why did we not find a second critical point corresponding to the minimum? The reason is that there is no specific function that achieves minimal entropy. As functions place more probability density on the two points $x = \mu + \sigma$ and $x = \mu - \sigma$, and place less probability density on all other values of x, they lose entropy while maintaining the desired variance. However, any function placing exactly zero mass on all but two points does not integrate to one, and is not a valid probability distribution. There thus is no single minimal entropy probability distribution function, much as there is no single minimal positive real number. Instead, we can say that there is a sequence of probability distributions converging toward putting mass only on these two points. This degenerate scenario may be

described as a mixture of Dirac distributions. Because Dirac distributions are not described by a single probability distribution function, no Dirac or mixture of Dirac distribution corresponds to a single specific point in function space. These distributions are thus invisible to our method of solving for a specific point where the functional derivatives are zero. This is a limitation of the method. Distributions such as the Dirac must be found by other methods, such as guessing the solution and then proving that it is correct.

19.4.3 Continuous Latent Variables

When our graphical model contains continuous latent variables, we may still perform variational inference and learning by maximizing $\mathcal{L}$. However, we must now use calculus of variations when maximizing $\mathcal{L}$ with respect to $q(\boldsymbol{h} \mid \boldsymbol{v})$.

In most cases, practitioners need not solve any calculus of variations problems themselves. Instead, there is a general equation for the mean field fixed point updates. If we make the mean field approximation

$$q(\boldsymbol{h} \mid \boldsymbol{v}) = \prod_i q(h_i \mid \boldsymbol{v}), \tag{19.55}$$

and fix $q(h_j \mid \boldsymbol{v})$ for all $j \neq i$, then the optimal $q(h_i \mid \boldsymbol{v})$ may be obtained by normalizing the unnormalized distribution

$$\tilde{q}(h_i \mid \boldsymbol{v}) = \exp\left(\mathbb{E}_{\mathbf{h}_{-i} \sim q(\mathbf{h}_{-i} \mid \boldsymbol{v})} \log \tilde{p}(\boldsymbol{v}, \boldsymbol{h})\right) \tag{19.56}$$

so long as p does not assign 0 probability to any joint configuration of variables. Carrying out the expectation inside the equation will yield the correct functional form of $q(h_i \mid \boldsymbol{v})$. It is only necessary to derive functional forms of q directly using calculus of variations if one wishes to develop a new form of variational learning; equation 19.56 yields the mean field approximation for any probabilistic model.

Equation 19.56 is a fixed point equation, designed to be iteratively applied for each value of i repeatedly until convergence. However, it also tells us more than that. It tells us the functional form that the optimal solution will take, whether we arrive there by fixed point equations or not. This means we can take the functional form from that equation but regard some of the values that appear in it as parameters, that we can optimize with any optimization algorithm we like.

As an example, consider a very simple probabilistic model, with latent variables $\boldsymbol{h} \in \mathbb{R}^2$ and just one visible variable, v. Suppose that $p(\boldsymbol{h}) = \mathcal{N}(\boldsymbol{h}; 0, \boldsymbol{I})$ and $p(v \mid \boldsymbol{h}) = \mathcal{N}(v; \boldsymbol{w}^\top \boldsymbol{h}; 1)$. We could actually simplify this model by integrating out $\boldsymbol{h}$; the result is just a Gaussian distribution over v. The model itself is not

interesting; we have constructed it only to provide a simple demonstration of how calculus of variations may be applied to probabilistic modeling.

The true posterior is given, up to a normalizing constant, by

$$p(\boldsymbol{h} \mid \boldsymbol{v}) \tag{19.57}$$
$$\propto p(\boldsymbol{h}, \boldsymbol{v}) \tag{19.58}$$
$$= p(h_1)p(h_2)p(\boldsymbol{v} \mid \boldsymbol{h}) \tag{19.59}$$
$$\propto \exp\left(-\frac{1}{2}\left[h_1^2 + h_2^2 + (v - h_1 w_1 - h_2 w_2)^2\right]\right) \tag{19.60}$$
$$= \exp\left(-\frac{1}{2}\left[h_1^2 + h_2^2 + v^2 + h_1^2 w_1^2 + h_2^2 w_2^2 - 2v h_1 w_1 - 2v h_2 w_2 + 2h_1 w_1 h_2 w_2\right]\right). \tag{19.61}$$

Due to the presence of the terms multiplying h_1 and h_2 together, we can see that the true posterior does not factorize over h_1 and h_2.

Applying equation 19.56, we find that

$$\tilde{q}(h_1 \mid \boldsymbol{v}) \tag{19.62}$$
$$= \exp\left(\mathbb{E}_{\mathrm{h}_2 \sim q(\mathrm{h}_2 \mid \boldsymbol{v})} \log \tilde{p}(\boldsymbol{v}, \boldsymbol{h})\right) \tag{19.63}$$
$$= \exp\left(-\frac{1}{2}\mathbb{E}_{\mathrm{h}_2 \sim q(\mathrm{h}_2 \mid \boldsymbol{v})}\left[h_1^2 + h_2^2 + v^2 + h_1^2 w_1^2 + h_2^2 w_2^2\right.\right. \tag{19.64}$$
$$\left.\left. - 2v h_1 w_1 - 2v h_2 w_2 + 2h_1 w_1 h_2 w_2\right]\right). \tag{19.65}$$

From this, we can see that there are effectively only two values we need to obtain from $q(h_2 \mid \boldsymbol{v})$: $\mathbb{E}_{\mathrm{h}_2 \sim q(\mathrm{h} \mid \boldsymbol{v})}[h_2]$ and $\mathbb{E}_{\mathrm{h}_2 \sim q(\mathrm{h} \mid \boldsymbol{v})}[h_2^2]$. Writing these as $\langle h_2 \rangle$ and $\langle h_2^2 \rangle$, we obtain

$$\tilde{q}(h_1 \mid \boldsymbol{v}) = \exp\left(-\frac{1}{2}\left[h_1^2 + \langle h_2^2 \rangle + v^2 + h_1^2 w_1^2 + \langle h_2^2 \rangle w_2^2\right.\right. \tag{19.66}$$
$$\left.\left. - 2v h_1 w_1 - 2v \langle h_2 \rangle w_2 + 2h_1 w_1 \langle h_2 \rangle w_2\right]\right). \tag{19.67}$$

From this, we can see that $\tilde{q}$ has the functional form of a Gaussian. We can thus conclude $q(\boldsymbol{h} \mid \boldsymbol{v}) = \mathcal{N}(\boldsymbol{h}; \boldsymbol{\mu}, \boldsymbol{\beta}^{-1})$ where $\boldsymbol{\mu}$ and diagonal $\boldsymbol{\beta}$ are variational parameters that we can optimize using any technique we choose. It is important to recall that we did not ever assume that q would be Gaussian; its Gaussian form was derived automatically by using calculus of variations to maximize q with

respect to $\mathcal{L}$. Using the same approach on a different model could yield a different functional form of q.

This was of course, just a small case constructed for demonstration purposes. For examples of real applications of variational learning with continuous variables in the context of deep learning, see Goodfellow *et al.* (2013d).

19.4.4 Interactions between Learning and Inference

Using approximate inference as part of a learning algorithm affects the learning process, and this in turn affects the accuracy of the inference algorithm.

Specifically, the training algorithm tends to adapt the model in a way that makes the approximating assumptions underlying the approximate inference algorithm become more true. When training the parameters, variational learning increases

$$\mathbb{E}_{\mathbf{h}\sim q} \log p(\boldsymbol{v}, \boldsymbol{h}). \tag{19.68}$$

For a specific $\boldsymbol{v}$, this increases $p(\boldsymbol{h} \mid \boldsymbol{v})$ for values of $\boldsymbol{h}$ that have high probability under $q(\boldsymbol{h} \mid \boldsymbol{v})$ and decreases $p(\boldsymbol{h} \mid \boldsymbol{v})$ for values of $\boldsymbol{h}$ that have low probability under $q(\boldsymbol{h} \mid \boldsymbol{v})$.

This behavior causes our approximating assumptions to become self-fulfilling prophecies. If we train the model with a unimodal approximate posterior, we will obtain a model with a true posterior that is far closer to unimodal than we would have obtained by training the model with exact inference.

Computing the true amount of harm imposed on a model by a variational approximation is thus very difficult. There exist several methods for estimating $\log p(\boldsymbol{v})$. We often estimate $\log p(\boldsymbol{v}; \boldsymbol{\theta})$ after training the model, and find that the gap with $\mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}, q)$ is small. From this, we can conclude that our variational approximation is accurate for the specific value of $\boldsymbol{\theta}$ that we obtained from the learning process. We should not conclude that our variational approximation is accurate in general or that the variational approximation did little harm to the learning process. To measure the true amount of harm induced by the variational approximation, we would need to know $\boldsymbol{\theta}^* = \max_{\boldsymbol{\theta}} \log p(\boldsymbol{v}; \boldsymbol{\theta})$. It is possible for $\mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}, q) \approx \log p(\boldsymbol{v}; \boldsymbol{\theta})$ and $\log p(\boldsymbol{v}; \boldsymbol{\theta}) \ll \log p(\boldsymbol{v}; \boldsymbol{\theta}^*)$ to hold simultaneously. If $\max_q \mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}^*, q) \ll \log p(\boldsymbol{v}; \boldsymbol{\theta}^*)$, because $\boldsymbol{\theta}^*$ induces too complicated of a posterior distribution for our q family to capture, then the learning process will never approach $\boldsymbol{\theta}^*$. Such a problem is very difficult to detect, because we can only know for sure that it happened if we have a superior learning algorithm that can find $\boldsymbol{\theta}^*$ for comparison.

19.5 Learned Approximate Inference

We have seen that inference can be thought of as an optimization procedure that increases the value of a function $\mathcal{L}$. Explicitly performing optimization via iterative procedures such as fixed point equations or gradient-based optimization is often very expensive and time-consuming. Many approaches to inference avoid this expense by learning to perform approximate inference. Specifically, we can think of the optimization process as a function f that maps an input $\boldsymbol{v}$ to an approximate distribution $q^* = \arg\max_q \mathcal{L}(\boldsymbol{v}, q)$. Once we think of the multi-step iterative optimization process as just being a function, we can approximate it with a neural network that implements an approximation $\hat{f}(\boldsymbol{v}; \boldsymbol{\theta})$.

19.5.1 Wake-Sleep

One of the main difficulties with training a model to infer $\boldsymbol{h}$ from $\boldsymbol{v}$ is that we do not have a supervised training set with which to train the model. Given a $\boldsymbol{v}$, we do not know the appropriate $\boldsymbol{h}$. The mapping from $\boldsymbol{v}$ to $\boldsymbol{h}$ depends on the choice of model family, and evolves throughout the learning process as $\boldsymbol{\theta}$ changes. The wake-sleep algorithm (Hinton *et al.*, 1995b; Frey *et al.*, 1996) resolves this problem by drawing samples of both $\boldsymbol{h}$ and $\boldsymbol{v}$ from the model distribution. For example, in a directed model, this can be done cheaply by performing ancestral sampling beginning at $\boldsymbol{h}$ and ending at $\boldsymbol{v}$. The inference network can then be trained to perform the reverse mapping: predicting which $\boldsymbol{h}$ caused the present $\boldsymbol{v}$. The main drawback to this approach is that we will only be able to train the inference network on values of $\boldsymbol{v}$ that have high probability under the model. Early in learning, the model distribution will not resemble the data distribution, so the inference network will not have an opportunity to learn on samples that resemble data.

In section 18.2 we saw that one possible explanation for the role of dream sleep in human beings and animals is that dreams could provide the negative phase samples that Monte Carlo training algorithms use to approximate the negative gradient of the log partition function of undirected models. Another possible explanation for biological dreaming is that it is providing samples from $p(\boldsymbol{h}, \boldsymbol{v})$ which can be used to train an inference network to predict $\boldsymbol{h}$ given $\boldsymbol{v}$. In some senses, this explanation is more satisfying than the partition function explanation. Monte Carlo algorithms generally do not perform well if they are run using only the positive phase of the gradient for several steps then with only the negative phase of the gradient for several steps. Human beings and animals are usually awake for several consecutive hours then asleep for several consecutive hours. It is

not readily apparent how this schedule could support Monte Carlo training of an undirected model. Learning algorithms based on maximizing $\mathcal{L}$ can be run with prolonged periods of improving q and prolonged periods of improving $\boldsymbol{\theta}$, however. If the role of biological dreaming is to train networks for predicting q, then this explains how animals are able to remain awake for several hours (the longer they are awake, the greater the gap between $\mathcal{L}$ and $\log p(\boldsymbol{v})$, but $\mathcal{L}$ will remain a lower bound) and to remain asleep for several hours (the generative model itself is not modified during sleep) without damaging their internal models. Of course, these ideas are purely speculative, and there is no hard evidence to suggest that dreaming accomplishes either of these goals. Dreaming may also serve reinforcement learning rather than probabilistic modeling, by sampling synthetic experiences from the animal's transition model, on which to train the animal's policy. Or sleep may serve some other purpose not yet anticipated by the machine learning community.

19.5.2 Other Forms of Learned Inference

This strategy of learned approximate inference has also been applied to other models. Salakhutdinov and Larochelle (2010) showed that a single pass in a learned inference network could yield faster inference than iterating the mean field fixed point equations in a DBM. The training procedure is based on running the inference network, then applying one step of mean field to improve its estimates, and training the inference network to output this refined estimate instead of its original estimate.

We have already seen in section 14.8 that the predictive sparse decomposition model trains a shallow encoder network to predict a sparse code for the input. This can be seen as a hybrid between an autoencoder and sparse coding. It is possible to devise probabilistic semantics for the model, under which the encoder may be viewed as performing learned approximate MAP inference. Due to its shallow encoder, PSD is not able to implement the kind of competition between units that we have seen in mean field inference. However, that problem can be remedied by training a deep encoder to perform learned approximate inference, as in the ISTA technique (Gregor and LeCun, 2010b).

Learned approximate inference has recently become one of the dominant approaches to generative modeling, in the form of the variational autoencoder (Kingma, 2013; Rezende *et al.*, 2014). In this elegant approach, there is no need to construct explicit targets for the inference network. Instead, the inference network is simply used to define $\mathcal{L}$, and then the parameters of the inference network are adapted to increase $\mathcal{L}$. This model is described in depth later, in section 20.10.3.

Using approximate inference, it is possible to train and use a wide variety of models. Many of these models are described in the next chapter.

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 of 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 can not easily be 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 multi-layer 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 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 backwards 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 have briefly described RBMs previously, 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.

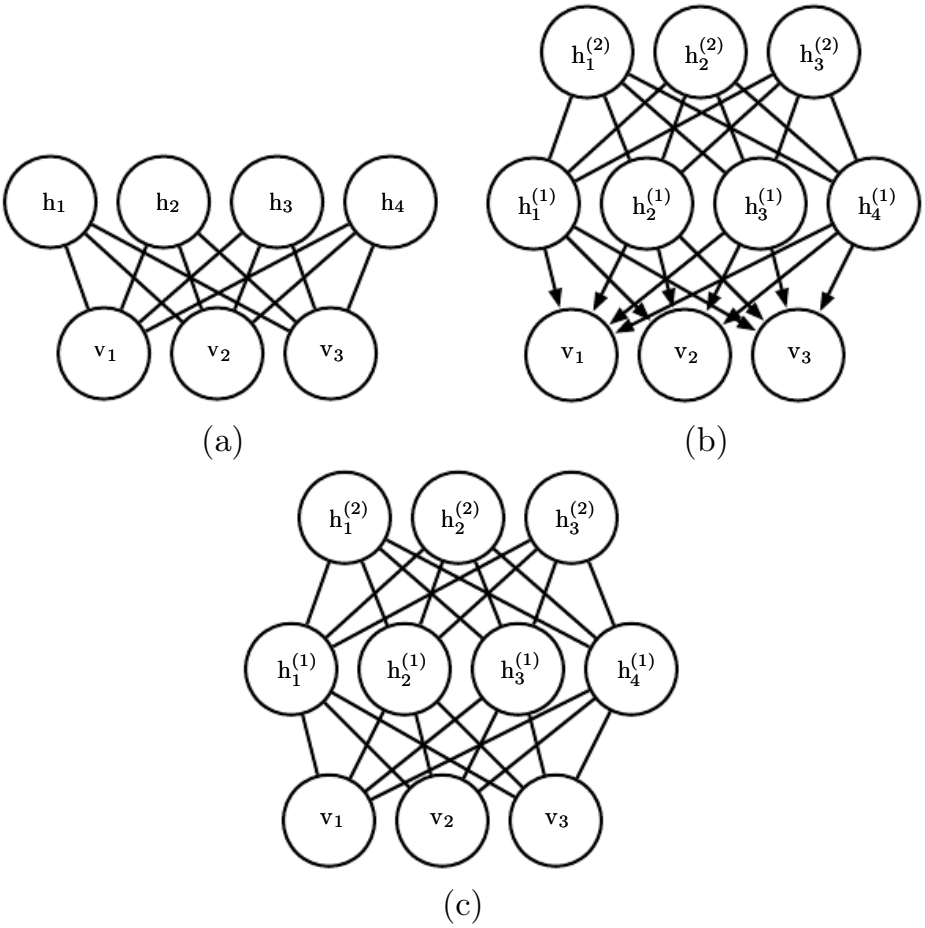

Figure 20.1: Examples of models that may be built with restricted Boltzmann machines. *(a)*The restricted Boltzmann machine itself is an undirected graphical model based on a bipartite graph, with visible units in one part of the graph and hidden units in the other part. There are no connections among 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 there are connections between hidden units that are in separate layers. All of 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 begin with the binary version of the restricted Boltzmann machine, but as 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 very special property that its conditional distributions $P(\mathbf{h} \mid \mathbf{v})$ and $P(\mathbf{v} \mid \mathbf{h})$ are factorial and relatively simple to compute and to 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 non-convolutional 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 due to the explaining away effect within each directed layer, and due to 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 not just confronting 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:

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

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

After initializing this MLP with the weights and biases learned via generative training of the DBN, we may 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 does not propagate any information downward or sideways. The DBN graphical model has explaining away interactions between all of the hidden units within the same layer as well as 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 "deep belief network" 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 "deep belief network" may also cause some confusion because the term "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.

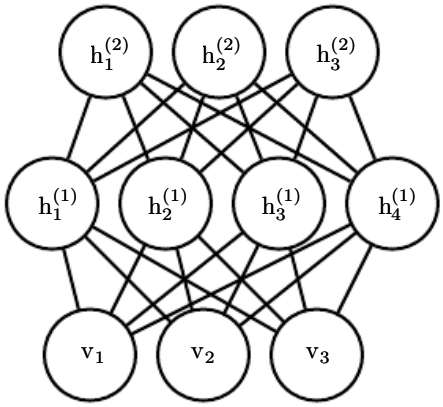

Figure 20.2: The graphical model for a deep Boltzmann machine with one visible layer (bottom) and two hidden layers. Connections are only between units in neighboring layers. There are no intralayer layer connections.

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 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:

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

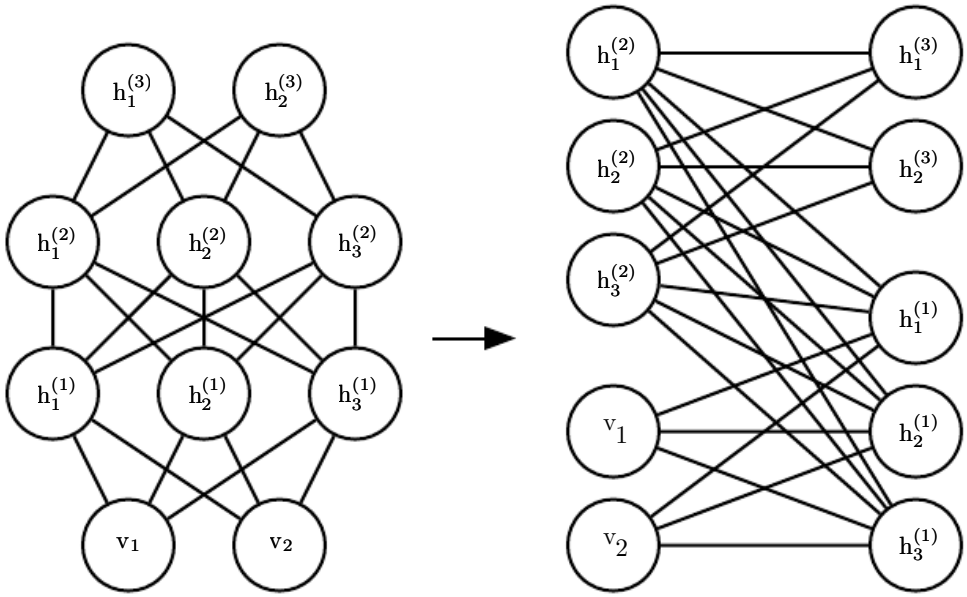

Figure 20.3: A deep Boltzmann machine, re-arranged to reveal its bipartite graph structure.

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 both 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:

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

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

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 of the visible units to be updated in one block and all of 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 of 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. Due to 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 $Q(\boldsymbol{h})$ may be used to obtain a variational lower bound on the log-likelihood. This heuristic procedure therefore allows us to obtain such a bound. However, the bound is not explicitly optimized in any way, so the bound 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 of 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 actually 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 due due to 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. However, unlike the DBN, 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 approxi-

mating 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 of the even layers simultaneously and then to update all of 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 non-variational version of stochastic maximum likelihood algorithm was discussed earlier, 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 varient 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 of 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 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 of 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

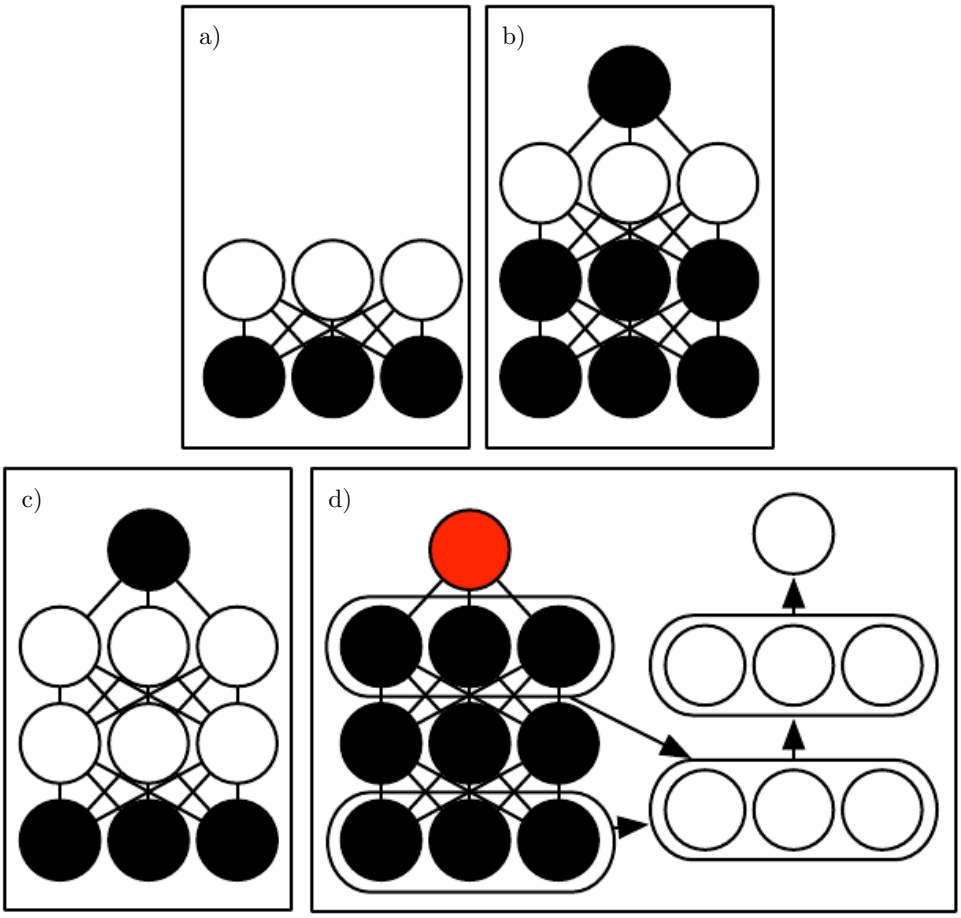

Figure 20.4: The deep Boltzmann machine training procedure used to classify the MNIST dataset (Salakhutdinov and Hinton, 2009a; Srivastava *et al.*, 2014). *(a)*Train an RBM by using CD to approximately maximize $\log P(\boldsymbol{v})$. *(b)*Train a second RBM that models $\boldsymbol{h}^{(1)}$ and 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).

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 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 in order 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 he 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 of 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 allows 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-

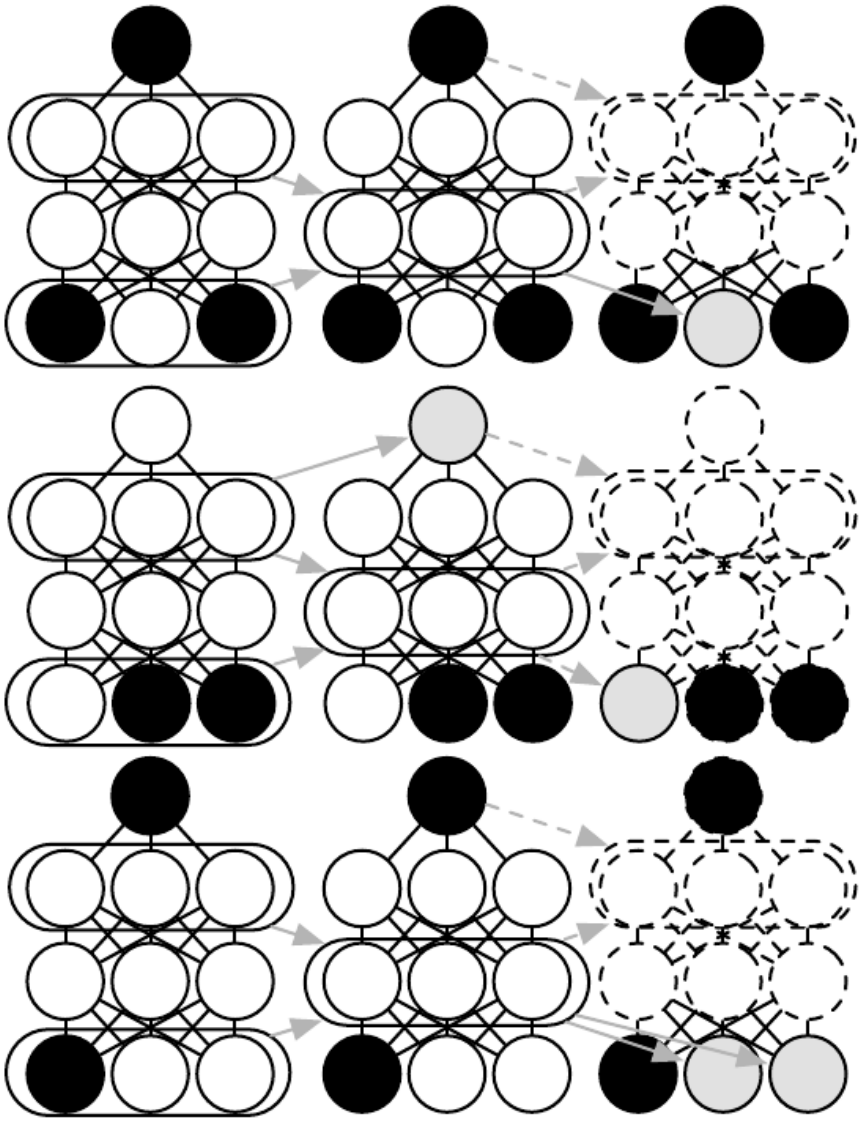

Figure 20.5: An illustration of the multi-prediction training process for a deep Boltzmann machine. Each row indicates a different example within a minibatch for the same training step. Each column represents a time step within the mean field inference process. For each example, we sample a subset of 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).

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 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 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. However, 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 of 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. However, we still have a choice 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 or not to include this bias term does not affect the family of distributions 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 either be 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 non-diagonal 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 t-distribution (mPoT) model and the spike and slab RBM (ssRBM).

[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."

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.

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:

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

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:

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

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 non-diagonal and that $\boldsymbol{W}$ is the weight matrix associated with the Gaussian RBM modeling the

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

conditional means. It is difficult to train the mcRBM via contrastive divergence or persistent contrastive divergence because of its non-diagonal 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.

Mean-Product of Student's t-distributions The mean-product of Student's 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 non-diagonal 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 non-diagonal covariance. Learning in the mPoT model—again, like the mcRBM—is complicated by the inability to sample from the non-diagonal 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:

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

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

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 non-negative 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, it is relatively straightforward to derive the ssRBM conditional distributions. 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:

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

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

where $\boldsymbol{C}^{\text{ss}}_{\boldsymbol{x}|\boldsymbol{h}} = \left(\boldsymbol{\Lambda} + \sum_i \boldsymbol{\Phi}_i h_i - \sum_i \alpha_i^{-1} h_i \boldsymbol{W}_{:,i} \boldsymbol{W}_{:,i}^\top\right)^{-1}$. The last equality holds only if the covariance matrix $\boldsymbol{C}^{\text{ss}}_{\boldsymbol{x}|\boldsymbol{h}}$ 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 very 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 requires 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 seen 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 of 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 spatial sizes. For Boltzmann machines, it is difficult to change the input size for a variety of 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 in order to obtain a fixed-size representation of a variable-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 variable-size 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% 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% in each direction to 3×3, then the mutual exclusivity constraint now specifies that each of these edges may only appear 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 variable-sized input images so 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, then there are fewer hidden units than visible units, and the visible units at the boundary of the image are not modeled

[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.

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 are 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