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. In chapter 20, we 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.

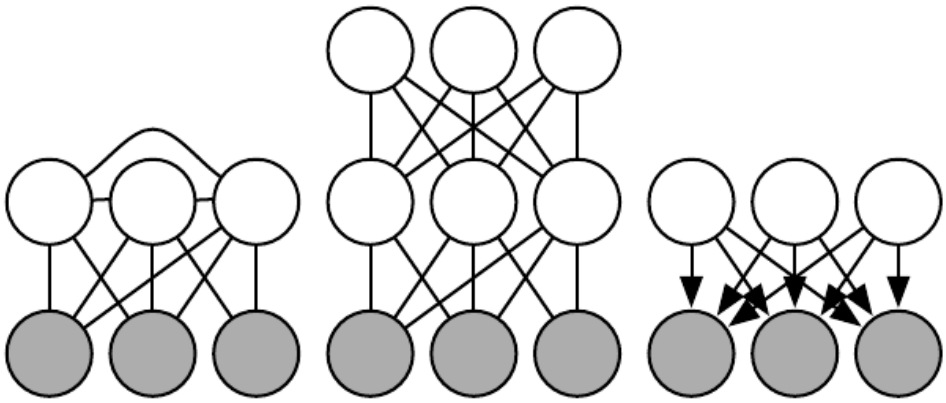

Figure 19.1: Intractable inference problems in deep learning are usually the result of interactions between latent variables in a structured graphical model. These interactions can be due to edges directly connecting one latent variable to another or longer paths that are activated when the child of 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 because of large cliques of latent variables. *(Center)*A deep Boltzmann machine, organized into layers of variables without intralayer connections, still has an intractable posterior distribution because of 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 coparents. 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 because of 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 nonnegative, 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 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 that the optimization is allowed to search over or by using an imperfect optimization procedure that may not completely maximize $\mathcal{L}$ but may 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 as 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. 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 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 the $\boldsymbol{h}$ vectors in the training set into a matrix $\boldsymbol{H}$, and concatenate all 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 $\boldsymbol{W}$, 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 subproblems are convex. In fact, the minimization with respect to $\boldsymbol{W}$ is just a linear regression problem. Minimization of J with respect to both arguments, however, 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 enables us to make large learning steps with a fixed q and that learning algorithms based on MAP inference enable 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. With 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 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. With 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 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:

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

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

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

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

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

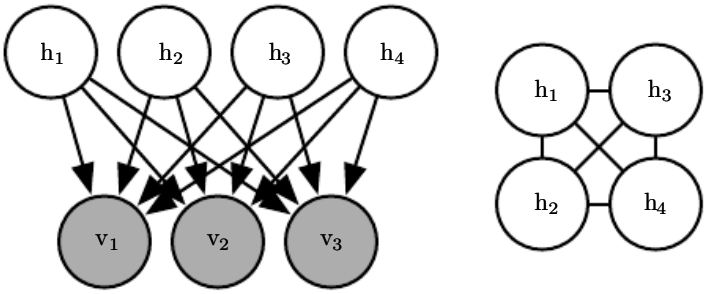

Figure 19.2: The graph structure of a binary sparse coding model with four hidden units. *(Left)*The graph structure of $p(\boldsymbol{h}, \boldsymbol{v})$. Note that the edges are directed, and that every two hidden units are coparents of every visible unit. *(Right)*The graph structure of $p(\boldsymbol{h}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

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

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

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. In a software implementation, however, 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 $\nabla_{\boldsymbol{h}} \mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}, \hat{\boldsymbol{h}}) = \mathbf{0}$. 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:

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

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

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

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

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

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

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

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

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 multimodal 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 will 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:

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

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, which 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 the values in a small step in that direction. This approach is no longer guaranteed to increase $\mathcal{L}$ at each step, but it 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 what 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 $p(x)$ integrate to 1. Also, the entropy should increase 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 rearranging 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 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 as long as the constraints are satisfied. To satisfy all 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 no specific function 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. Thus there 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 can 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

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

as 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 $q(h_i \mid \boldsymbol{v})$. Deriving functional forms of q directly using calculus of variations is only necessary 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, which we can optimize with any optimization algorithm we like.

As an example, consider a 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 can be applied to probabilistic modeling.

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

$$p(\boldsymbol{h} \mid \boldsymbol{v}) \tag{19.57}$$

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

Because of 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, which 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 multistep 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 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.