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 chapters 13, 18 and 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 sampling. Some of these models are structured probabilistic models described in terms of graphs and factors, as described in chapter 13. Others can not easily be described in terms of factors, but represent probability distributions nonetheless.

20.1 Boltzmann Machines

Boltzmann machines were originally introduced in Ackley *et al.* (1985) as a general "connectionist" approach to learning arbitrary probability distributions over binary vectors. Boltzmann Machines form the basis of a large number of popular variants. Indeed, these variants have long ago surpassed the popularity of the original and most general incarnation of the Boltzmann machine. In this section we briefly introduce the general Boltzmann machine and discuss the issues that come up when trying to train and perform inference in the model.

We define our Boltzmann machine over a d-dimensional binary random vector $\mathbf{x} \in \{0, 1\}^d$. The Boltzmann machine is an energy-based model[1], meaning we define the joint probability distribution over the model variable 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

[1] For a general discussion of energy based models see Sec. 13.2.4

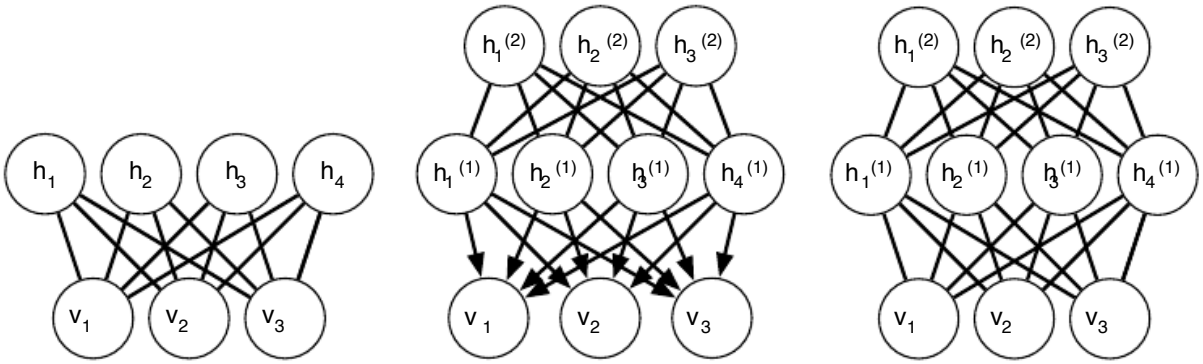

Figure 20.1: Examples of models that may be built with restricted Boltzmann machines. *a)* The restricted Boltzmann machine itself is an undirected graphical model based on a bipartite graph. There are no connections among the visible units, nor any connections among the hidden units. Typically every 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 intra-layer 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. Note that we could also represent the deep belief network with a completely undirected graph, but it would need intra-layer 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 intra-layer 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.

that the $\sum_{\boldsymbol{x}} P(\boldsymbol{x}) = 1$. 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 $\boldsymbol{b}$ are the offsets for each $\boldsymbol{x}$.

In the general setting of the Boltzmann machine, we could consider that the goal is that we are given a set of observations, each of which are d-dimensional and that we are to use the joint probability distribution given in Eq. 20.1 describes the joint probability distribution over the observed variables (also called *visible units*). 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 limits the model to 2nd-order interactions.

In the spirit of the "connectionist" approach to density modeling that originally inspired the Boltzmann machine, it is interesting to consider the case where not all the variables are observed. In this case, the non-observed variables, or *latent* variables can act similarly to hidden units in a multi-layer perceptron and model higher-order interactions among the visible units.

Formally, we decompose the units into two subsets: the visible units $\boldsymbol{x}_v$ and the latent (or hidden) units $\boldsymbol{x}_h$. Without loss of generality, we can re-express the energy function decomposing $\boldsymbol{x}$ into subsets $\boldsymbol{x}_v$ and $\boldsymbol{x}_h$:

$$E(\boldsymbol{x}_v, \boldsymbol{x}_h) = -\boldsymbol{x}_v^\top \boldsymbol{R}\boldsymbol{x}_v - \boldsymbol{x}_v^\top \boldsymbol{W}\boldsymbol{x}_h - \boldsymbol{x}_h^\top \boldsymbol{S}\boldsymbol{x}_h - \boldsymbol{b}^\top \boldsymbol{x}_v - \boldsymbol{c}^\top \boldsymbol{x}_h, \tag{20.3}$$

Boltzmann Machine Learning As a probabilistic model, it is natural to consider maximum likelihood as the learning paradigm for Boltzmann machines. According to the ML paradigm, we are interested in choosing the parameters that (locally) maximize the probability of the visible units over a dataset.

Consider a dataset of n examples $\boldsymbol{X}_v = [\boldsymbol{x}_v^{(1)}, \ldots, \boldsymbol{x}_v^{(t)}, \ldots, \boldsymbol{x}_v^{n}]$. Our goal is to maximize the likelihood of this dataset under the Boltzmann machine. Assuming the data is i.i.d, this amounts to maximizing the following:

$$\ell(\boldsymbol{\theta}) = \log P(\boldsymbol{X}_v) = \sum_{t=1}^{n} \log P(\boldsymbol{x}_v^{(t)}). \tag{20.4}$$

Of course, our Boltzmann machine does not explicitly parametrize a distribution over the visible units as $P(\boldsymbol{x}_v^{(t)})$, instead, as given in Eq. 20.3, it is parametrized via an energy function to joint probability distribution over $\boldsymbol{x}_v$ and $\boldsymbol{x}_h$, the hidden units. In order to recover $P(\boldsymbol{x}_v^{(t)})$, we need to *marginalize out* the influence of $\boldsymbol{x}_h$.

$$P(\boldsymbol{x}_v^{(t)}) = \sum_{\boldsymbol{x}_h} P(\boldsymbol{x}_v^{(t)}, \boldsymbol{x}_h^{(t)}) = \sum_{\boldsymbol{x}_h} \frac{1}{Z} \exp\left\{ -E(\boldsymbol{x}_v^{(t)}, \boldsymbol{x}_h^{(t)}) \right\}. \tag{20.5}$$

Combining Eqs. 20.4 and 20.5 gives us our objective function we wish to maximize. Unfortunately, because Z is a function of the model parameters, maximizing likelihood function is not amenable to analytically solution. Instead we will do as we do for the vast majority of deep learning models, we will follow the gradient of our objective function. The Boltzmann machine likelihood gradient is given by:

$$\frac{\partial}{\partial \boldsymbol{\theta}} \ell(\boldsymbol{\theta}) = \frac{\partial}{\partial \boldsymbol{\theta}} \sum_{t=1}^{n} \left[\frac{1}{Z} \log \sum_{\boldsymbol{x}_h} \exp \left\{ -E(\boldsymbol{x}_v^{(t)}, \boldsymbol{x}_h^{(t)}) \right\} \right] \Bigg) \tag{20.6}$$

$$= \sum_{t=1}^{n} \frac{\partial}{\partial \boldsymbol{\theta}} \left[\log \sum_{\boldsymbol{x}_h} \exp \Big\{ -E(\boldsymbol{x}_v^{(t)}, \boldsymbol{x}_h^{(t)}) \Big\} \right] - \frac{\partial Z}{\partial \boldsymbol{\theta}} \tag{20.7}$$

$$= \sum_{t=1}^{n} \left[\sum_{\boldsymbol{x}_h} \frac{\exp \left\{ -E(\boldsymbol{x}_v^{(t)}, \boldsymbol{x}_h^{(t)}) \right\}}{\sum_{\boldsymbol{x}_h} \exp \left\{ -E(\boldsymbol{x}_v^{(t)}, \boldsymbol{x}_h^{(t)}) \right\}} \frac{\partial}{\partial \boldsymbol{\theta}} E(\boldsymbol{x}_v^{(t)}, \boldsymbol{x}_h^{(t)}) \right] - \frac{\partial Z}{\partial \boldsymbol{\theta}} \tag{20.8}$$

20.2 Restricted Boltzmann Machines

Restricted Boltzmann machines are some of the most common building blocks of deep probabilistic models. They 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 Fig. 20.1 for some examples. In particular, Fig. 20.1a shows the graph structure of an 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.

TODO– review and pointers to other sections of the book This should be the main place where they are described in detail, earlier they are just an example of undirected models or an example of a feature learning algorithm.

TODO: please use lower-case letter names for scalars, none of this D and N stuff. do we even use these variable names anywhere, or do we just define them and never refer back to them? if they are never used, delete them, don't make the reader hold variables in their head for no payoff

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, we will consider the observed layer to consist of a set of D binary random variables which we refer to collectively with the vector $\mathbf{v}$, where the ith element, i.e. v_i is a binary random variable. We will refer to the latent or hidden layer of N binary random variables collectively as $\boldsymbol{h}$, with the jth random elements as h_j.

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

function:

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

Where $E(\boldsymbol{v}, \boldsymbol{h})$ is the energy function that parametrizes the relationship between the visible and hidden variables:

$$E(\boldsymbol{v}, \boldsymbol{h}) = -\boldsymbol{b}^\top \boldsymbol{v} - \boldsymbol{c}^\top \boldsymbol{h} - \boldsymbol{v}^\top \boldsymbol{W} \boldsymbol{h}, \tag{20.9}$$

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

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

For many undirected models, it is apparent from the definition of the partition function Z that the naive method of computing Z (exhaustively summing over all states) would be computationally intractable. However, it is still possible that a more cleverly designed algorithm could exploit regularities in the probability distribution to compute Z faster than the naive algorithm, so the exponential cost of the naive algorithm is not a guarantee of the partiion function's intractability. In the case of restricted Boltzmann machines, there is actually a hardness result, proven by Long and Servedio (2010).

20.2.1 Conditional Distributions

The intractable partition function Z, implies that the joint probability distribution is also intractable (in the sense that the normalized probability of a given joint configuration of $[\boldsymbol{v}, \boldsymbol{h}]$ is generally not available). However, due the bipartite graph structure, the restricted Boltzmann machine 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 sample from. Indeed, it is this property that has made the RBM a relatively popular model for a wide range of applications including image modeling (TODO CITE), speech processing (TODO CITE) and natural language processing (TODO CITE).

Deriving the conditional distributions from the joint distribution is straight-

forward.

$$
\begin{aligned}
p(\boldsymbol{h} \mid \boldsymbol{v}) &= \frac{p(\boldsymbol{h}, \boldsymbol{v})}{p(\boldsymbol{v})} \\
&= \frac{p(\boldsymbol{h}, \boldsymbol{v})}{p(\boldsymbol{v})} \\
&= \frac{1}{p(\boldsymbol{v})} \frac{1}{Z} \exp\Big\{ \boldsymbol{b}^\top \boldsymbol{v} + \boldsymbol{c}^\top \boldsymbol{h} + \boldsymbol{v}^\top \boldsymbol{W} \boldsymbol{h} \Big\} \\
&= \frac{1}{Z'} \exp\Big\{ \boldsymbol{c}^\top \boldsymbol{h} + \boldsymbol{v}^\top \boldsymbol{W} \boldsymbol{h} \Big\} \\
&= \frac{1}{Z'} \exp\left\{ \sum_{j=1}^{n} c_j h_j + \sum_{j=1}^{n} \boldsymbol{v}^\top \boldsymbol{W}_{:,j} \boldsymbol{h}_j \right\} \\
&= \frac{1}{Z'} \prod_{j=1}^{n} \exp\Big\{ c_j h_j + \boldsymbol{v}^\top \boldsymbol{W}_{:,j} \boldsymbol{h}_j \Big\}
\end{aligned}
$$

Since we are conditioning on the visible units $\mathbf{v}$, we can treat these as constants w.r.t. 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 wright 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.

$$
\begin{aligned}
P(h_j = 1 \mid \boldsymbol{v}) &= \frac{\tilde{P}(h_j = 1 \mid \boldsymbol{v})}{\tilde{P}(h_j = 0 \mid \boldsymbol{v}) + \tilde{P}(h_j = 1 \mid \boldsymbol{v})} \\
&= \frac{\exp\left\{ c_j + \boldsymbol{v}^\top \boldsymbol{W}_{:,j} \right\}}{\exp\{0\} + \exp\left\{ c_j + \boldsymbol{v}^\top \boldsymbol{W}_{:,j} \right\}} \\
&= \text{sigmoid}\left(c_j + \boldsymbol{v}^\top \boldsymbol{W}_{:,j} \right). \qquad (20.10)
\end{aligned}
$$

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

$$
P(\boldsymbol{h} \mid \boldsymbol{v}) = \prod_{j=1}^{n} \text{sigmoid}\left(c_j + \boldsymbol{v}^\top \boldsymbol{W}_{:,j} \right). \qquad (20.11)
$$

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

$$
P(\boldsymbol{v} \mid \boldsymbol{h}) = \prod_{i=1}^{d} \text{sigmoid}\left(b_i + \boldsymbol{W}_{i,:} \boldsymbol{h} \right). \qquad (20.12)
$$

20.2.2 RBM Gibbs Sampling

The factorial nature of these conditions is a very useful property of the RBM, and allows us to efficiently draw samples from the joint distribution via a block Gibbs sampling strategy (see section 14.1 for a more complete discussion of Gibbs sampling methods).

Block Gibbs sampling simply refers to the situation where in each step of Gibbs sampling, multiple variables (or a "block" of variables) are sampled jointly. In the case of the RBM, each iteration of block Gibbs sampling consists of two steps. **Step 1:** Sample $\boldsymbol{h}^{(l)} \sim P(\boldsymbol{h} \mid \boldsymbol{v}^{(l)})$. Due to the factorial nature of the conditionals, we can simultaneously and independently sample from all the elements of $\boldsymbol{h}^{(l)}$ given $\boldsymbol{v}^{(l)}$. **Step 2:** Sample $\boldsymbol{v}^{(l+1)} \sim P(\boldsymbol{v} \mid \boldsymbol{h}^{(l)})$. Again, the factorial nature of the conditional $P(\boldsymbol{v} \mid \boldsymbol{h}^{(l)})$ allows us can simultaneously and independently sample from all the elements of $\boldsymbol{v}^{(l+1)}$ given $\boldsymbol{h}^{(l)}$.

20.3 Training Restricted Boltzmann Machines

Despite the simplicity of the RBM conditionals, training these models is not without its complications. As a probabilistic model, a sensible inductive principle for estimating the model parameters is maximum likelihood – though other possibilities are certainly possible Marlin *et al.* (2010) and will be discussed later in Sec. 20.3.3. In the following we derive the maximum likelihood gradient with respect to the model parameters.

Let us consider that we have a batch (or minibatch) of n examples taken from an i.i.d dataset (independently and identically distributed examples) $\{\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(t)}, \ldots, \boldsymbol{v}^{(n)}\}$. The log-likelihood under the RBM with parameters $\boldsymbol{b}$ (visible unit biases), $\boldsymbol{c}$ (hidden unit biases) and $\boldsymbol{W}$ (interaction weights) is given by:

$$
\begin{aligned}
\ell(\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{c}) &= \sum_{t=1}^{n} \log P(\boldsymbol{v}^{(t)}) \\
&= \sum_{t=1}^{n} \log \sum_{\boldsymbol{h}} P(\boldsymbol{v}_{n,:}^{(t)}, \boldsymbol{h}) \\
&= \sum_{t=1}^{n} \log \sum_{\boldsymbol{h}} \exp\left\{-E(\boldsymbol{v}^{(t)}, \boldsymbol{h})\right\} \Bigg) - n \log Z \\
&= \sum_{t=1}^{n} \log \sum_{\boldsymbol{h}} \exp\left\{-E(\boldsymbol{v}^{(t)}, \boldsymbol{h})\right\} \Bigg) - n \log \sum_{\boldsymbol{v},\boldsymbol{h}} \exp\{-E(\boldsymbol{v}, \boldsymbol{h})\}
\end{aligned}
\tag{20.13}
$$

In the last line of the equation above, we have used the definition of the partition function.

To maximize the likelihood of the data under the restricted Boltzmann machine, we consider the gradient of the likelihood with respect to the model parameters, which we will refer to collectively as $\boldsymbol{\theta} = \{\boldsymbol{b}, \boldsymbol{c}, \boldsymbol{W}\}$:

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}} \ell(\boldsymbol{\theta}) &= \nabla_{\boldsymbol{\theta}} \quad \sum_{t=1}^{n} \log \sum_{\boldsymbol{h}} \exp\Big\{-E(\boldsymbol{v}^{(t)}, \boldsymbol{h})\Big\}\Bigg) - n \frac{\partial}{\partial \theta} \log \sum_{\boldsymbol{v},\boldsymbol{h}} \exp\{-E(\boldsymbol{v}, \boldsymbol{h})\} \\
&= \sum_{t=1}^{n} \frac{\sum_{\boldsymbol{h}} \exp\{-E(\boldsymbol{v}^{(t)}, \boldsymbol{h})\} \nabla_{\boldsymbol{\theta}} - E(\boldsymbol{v}^{(t)}, \boldsymbol{h})}{\sum_{\boldsymbol{h}} \exp\big\{-E(\boldsymbol{v}^{(t)}, \boldsymbol{h})\big\}} - n \frac{\sum_{\boldsymbol{v},\boldsymbol{h}} \exp\{-E(\boldsymbol{v}, \boldsymbol{h})\} \nabla_{\boldsymbol{\theta}} - E(\boldsymbol{v}, \boldsymbol{h})}{\sum_{\boldsymbol{v},\boldsymbol{h}} \exp\{-E(\boldsymbol{v}, \boldsymbol{h})\}} \\
&= \sum_{t=1}^{n} \mathbb{E}_{P(\boldsymbol{h}|\boldsymbol{v}^{(t)})} \Big[\nabla_{\boldsymbol{\theta}} - E(\boldsymbol{v}^{(t)}, \boldsymbol{h})\Big] - n \mathbb{E}_{P(\boldsymbol{v},\boldsymbol{h})} [\nabla_{\boldsymbol{\theta}} - E(\boldsymbol{v}, \boldsymbol{h})] \qquad (20.14)
\end{aligned}$$

As we can see from Eq. 20.14, the gradient of the log likelihood is specified as the difference between two expectations of the gradient of the energy function, The first expectation (the *data term*) is with respect to the product of the empirical distribution over the data, $P(v) = 1/n \sum_{t=1}^{n} \delta(x - \boldsymbol{v}^{(t)})$ [2] and the conditional distribution $P(\boldsymbol{h} \mid \boldsymbol{v}^{(t)})$. The second expectation (the *model term*) is with respect to the joint model distribution $P(\boldsymbol{v}, \boldsymbol{h})$.

This difference between a data-driven term and a model-driven term is not unique to RBMs, as discussed in some detail in Sec. 18.2, this is a general feature of the maximum likelihood gradient for all undirected models.

We can complete the derivation of log-likelihood gradient by expanding the term: $\nabla_{\boldsymbol{\theta}} - E(\boldsymbol{v}, \boldsymbol{h})$. We will consider first the gradient of the negative energy function of $\boldsymbol{W}$.

(20.15)

$$\begin{aligned}
nabla_{\boldsymbol{W}} - E(\boldsymbol{v}, \boldsymbol{h}) &= \frac{\partial}{\partial \boldsymbol{W}} \Big(\boldsymbol{b}^{\top} \boldsymbol{v} + \boldsymbol{c}^{\top} \boldsymbol{h} + \boldsymbol{v}^{\top} \boldsymbol{W} \boldsymbol{h}\Big) \\
&= \boldsymbol{h} \boldsymbol{v}^{\top} \qquad (20.16)
\end{aligned}$$

The gradients with respect to $\boldsymbol{b}$ and $\boldsymbol{c}$ are similarly derived:

$$\nabla_{\boldsymbol{b}} - E(\boldsymbol{v}, \boldsymbol{h}) = \boldsymbol{v}, \nabla_{\boldsymbol{c}} - E(\boldsymbol{v}, \boldsymbol{h}) = \boldsymbol{h} \qquad (20.17)$$

[2] As discussed in Sec. 3.10.5, we use the term empirical distribution to refer to a mixture over delta functions placed on training examples

Putting it all together we can the following equations for the gradients with respect to the RBM parameters and given n training examples:

$$\nabla_{\boldsymbol{W}} \ell(\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{c}) = \sum_{t=1}^{n} \hat{\boldsymbol{h}}^{(t)} \, \boldsymbol{v}^{(t)\top} - N\mathbb{E}_{P(\boldsymbol{v},\boldsymbol{h})} \left[\boldsymbol{h}\boldsymbol{v}^{\top}\right]$$

$$\nabla_{\boldsymbol{b}} \ell(\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{c}) = \sum_{t=1}^{n} \boldsymbol{v}^{(t)} - n\mathbb{E}_{P(\boldsymbol{v},\boldsymbol{h})}[\boldsymbol{v}]$$

$$\nabla_{\boldsymbol{c}} \ell(\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{c}) = \sum_{t=1}^{n} \hat{\boldsymbol{h}}^{(t)} - n\mathbb{E}_{P(\boldsymbol{v},\boldsymbol{h})}[\boldsymbol{h}]$$

where we have defined $\hat{\boldsymbol{h}}^{(t)}$ as

$$\hat{\boldsymbol{h}}^{(t)} = \mathbb{E}_{P(\boldsymbol{h}|\boldsymbol{v}^{(t)})}[\boldsymbol{h}] = \text{sigmoid}\Big(\boldsymbol{c} + \boldsymbol{v}^{(t)}\boldsymbol{W}\Big). \tag{20.18}$$

While we are able to write down these expressions for the log-likelihood gradient, unfortunately, in most situations of interest, we are not able to use them directly to calculate gradients. The problem is the expectations over the joint model distribution $P(\boldsymbol{v}, \boldsymbol{h})$. While we have conditional distributions $P(\boldsymbol{v} \mid \boldsymbol{h})$ and $P(\boldsymbol{h} \mid \boldsymbol{v})$ that are easy to work with, the RBM joint distribution is not amenable to analytic evaluation of the expectation $\mathbb{E}_{P(\boldsymbol{v},\boldsymbol{h})}[f(\boldsymbol{v}, \boldsymbol{h})]$.

This is bad news—it implies that in most cases it is impractical to compute the exact log-likelihood gradient. Fortunately, as discussed in Sec. 18.2, there are two widely used approximation strategies that have been applied to the training of RBM with some degree of success: contrastive divergence and stochastic maximum likelihood.

In the following sections we discuss two different strategies to approximate this gradient that have been applied to training the RBM. However, before getting into the actual training algorithms, it is worth considering what general approaches are available to us in approximating the log-likelihood gradient. As we mentioned, our problem stems from the expectation over the joint distribution $P(\boldsymbol{v}, \boldsymbol{h})$, but we know that we have access to factorial conditionals and that we can use these as the basis of a Gibbs sampling procedure to recover samples form the joint distribution (as discussed in Sec. 20.2.2). Thus, we can imagine using, for example, T MCMC samples from $P(\boldsymbol{v}, \boldsymbol{h})$ to form a Monte Carlo estimate of the expectations over the joint distribution:

$$\mathbb{E}_{P(\boldsymbol{v},\boldsymbol{h})}[f(\boldsymbol{v}, \boldsymbol{h})] \approx \frac{1}{T} \sum_{t=1}^{T} f(\boldsymbol{v}^{(t)}, \boldsymbol{h}^{(t)}). \tag{20.19}$$

There is a problem with this strategy that has to do with the initialization of the MCMC chain. MCMC chains typically require a burn-in period, where the chain

20.3.1 Contrastive Divergence Training of the RBM

As discussed in a more general context in Sec. 18.2, Contrastive divergence (CD) seeks to approximate the expectation over the joint distribution with samples drawn from short Gibbs sampling chains. CD deals with the typical requirement for an extended burn-in sample sequence by initializing these chains at the data points used in the data-dependent, conditional term. The result is a biased approximation of the log-likelihood gradient (Carreira-Perpiñan and Hinton, 2005; Bengio and Delalleau, 2009; Fischer and Igel, 2011), that never-the-less has been empirically shown to be effective. The constrastive divergence algorithm, as applied to RBMs, is given in Algorithm 20.1.

Algorithm 20.1 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 of $p(\boldsymbol{v}; \theta)$ to mix when initializedfrom p_{data}. Perhaps 1-20 to train an RBM on a small image patch.

while Not converged **do**

  Sample a minibatch of m examples from the training set $\{\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(m)}\}$.

  $\Delta_{\boldsymbol{W}} \leftarrow \frac{1}{m} \sum_{t=1}^{m} \boldsymbol{v}^{(t)} \hat{\boldsymbol{h}}^{(t)\top}$

  $\Delta_{\boldsymbol{b}} \leftarrow \frac{1}{m} \sum_{t=1}^{m} \boldsymbol{v}^{(t)}$

  $\Delta_{\boldsymbol{c}} \leftarrow \frac{1}{m} \sum_{t=1}^{m} \hat{\boldsymbol{h}}^{(t)}$

  for $t = 1$ to m **do**

    $\tilde{\boldsymbol{v}}^{(t)} \leftarrow \boldsymbol{v}^{(t)}$

  end for

  for $l = 1$ to k **do**

    for $t = 1$ to m **do**

      $\tilde{\boldsymbol{h}}^{(t)}$ sampled from $\prod_{j=1}^{n}$ sigmoid $\left(c_j + \tilde{\boldsymbol{v}}^{(t)\top} \boldsymbol{W}_{:,j}\right)$.

      $\tilde{\boldsymbol{v}}^{(t)}$ sampled from $\prod_{i=1}^{d}$ sigmoid $\left(b_i + \boldsymbol{W}_{i,:} \tilde{\boldsymbol{h}}^{(t)}\right)$.

    end for

  end for

  $\bar{\boldsymbol{h}}^{(t)} \leftarrow$ sigmoid $\left(\boldsymbol{c} + \tilde{\boldsymbol{v}}^{(t)\top} \boldsymbol{W}\right)$

  $\Delta \boldsymbol{W} \leftarrow \Delta \boldsymbol{W} - \frac{1}{m} \sum_{t=1}^{m} \tilde{\boldsymbol{v}}^{(t)} \bar{\boldsymbol{h}}^{(t)\top}$

  $\Delta_{\boldsymbol{b}} \leftarrow \Delta_{\boldsymbol{b}} - \frac{1}{m} \sum_{t=1}^{m} \tilde{\boldsymbol{v}}^{(t)}$

  $\Delta_{\boldsymbol{c}} \leftarrow \Delta_{\boldsymbol{b}} - \frac{1}{m} \sum_{t=1}^{m} \bar{\boldsymbol{h}}^{(t)}$

  $\boldsymbol{W} \leftarrow \boldsymbol{W} + \epsilon \Delta_{\boldsymbol{W}}$

  $\boldsymbol{b} \leftarrow \boldsymbol{b} + \epsilon \Delta_{\boldsymbol{b}}$

  $\boldsymbol{c} \leftarrow \boldsymbol{c} + \epsilon \Delta_{\boldsymbol{c}}$

end while

20.3.2 Stochastic Maximum Likelihood for the RBM

While contrastive divergence has been the most popular method of training RBMs, the *stochastic maximum likelihood* (SML) algorithm (Younes, 1998; Tieleman, 2008) is known to be a competitive alternative – especially if we are interested in recovering the best possible generative model (i.e. achieving the highest possible test set likelihood). As with CD, the general SML algorithm is described in Sec. 18.2. Here we are concerned with how to apply the algorithm to training an RBM.

In comparison to the CD algorithm, SML uses an alternative solution to the problem of how to approximate the partition function's contribution to the log-likelihood gradient. Instead of initializing the k-step MCMC chain with the current example from the training set, in SML we initialize the MCMC chain for training iteration s with the last state of the MCMC chain from the last training iteration $(s-1)$. Assuming that the gradient updates to the model parameters do not significantly change the model, the MCMC state of the last iteration should be close to the equilibrium distribution at iteration s – minimizing the number of "burn-in" MCMC steps needed to reach equilibrium at the current iteration. As with CD, in practice we often use just one Gibbs step between learning iterations. Algorithm 20.2 describes the SML algorithm as applied to RBMs.

TODO: include experimental examples, i.e. an RBM trained with CD on MNIST

20.3.3 Other Inductive Principles

TODO:Other inductive principles have been used to train RBMs. In this section we briefly discuss these.

20.4 Deep Belief Networks

Deep belief networks (DBNs) were one of the first successful non-convolutional architectures. 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, due to the vanishing and exploding gradient problems and the existence of plateaus, negative curvature, and suboptimal local minima that can arise in neural network objective functions. 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

Algorithm 20.2 The stochastic maximum likelihood / persistent contrastive divergence algorithm for training an RBM.

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 $p(\boldsymbol{v}, \boldsymbol{h}; \boldsymbol{\theta} + \epsilon \Delta_{\boldsymbol{\theta}})$ toburn in, starting from samples from $p(\boldsymbol{v}, \boldsymbol{h}; \boldsymbol{\theta})$. Perhaps 1 for RBM on a small image patch.
Initialize a set of m samples $\{\tilde{\boldsymbol{v}}^{(1)}, \ldots, \tilde{\boldsymbol{v}}^{(m)}\}$ 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 $\{\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(m)}\}$ from the training set.
  $\Delta_{\boldsymbol{W}} \leftarrow \frac{1}{m} \sum_{t=1}^{m} \hat{\boldsymbol{h}}^{(t)} \boldsymbol{v}^{(t)\top}$
  $\Delta_{\boldsymbol{b}} \leftarrow \frac{1}{m} \sum_{t=1}^{m} \boldsymbol{v}^{(t)}$
  $\Delta_{\boldsymbol{c}} \leftarrow \frac{1}{m} \sum_{t=1}^{m} \hat{\boldsymbol{h}}^{(t)}$
  for $l = 1$ to k **do**
    for $t = 1$ to m **do**
      $\tilde{\boldsymbol{h}}^{(t)}$ sampled from$\prod_{j=1}^{n}$ sigmoid $\left(c_j + \tilde{\boldsymbol{v}}^{(t)\top} \boldsymbol{W}_{:,j}\right)$.
      $\tilde{\boldsymbol{v}}^{(t)}$sampled from $\prod_{i=1}^{d}$sigmoid $\left(b_i + \boldsymbol{W}_{i,:} \tilde{\boldsymbol{h}}^{(t)}\right)$.
    end for
  end for
  $\Delta_{\boldsymbol{W}} \leftarrow \Delta_{\boldsymbol{W}} - \frac{1}{m} \sum_{t=1}^{m} \tilde{\boldsymbol{v}}^{(t)} \tilde{\boldsymbol{h}}^{(t)\top}$
  $\Delta_{\boldsymbol{b}} \leftarrow \Delta_{\boldsymbol{b}} - \frac{1}{m} \sum_{t=1}^{m} \tilde{\boldsymbol{v}}^{(t)}$
  $\Delta_{\boldsymbol{c}} \leftarrow \Delta_{\boldsymbol{b}} - \frac{1}{m} \sum_{t=1}^{m} \tilde{\boldsymbol{h}}^{(t)}$
  $\boldsymbol{W} \leftarrow \boldsymbol{W} + \epsilon \Delta_{\boldsymbol{W}}$
  $\boldsymbol{b} \leftarrow \boldsymbol{b} + \epsilon \Delta_{\boldsymbol{b}}$
  $\boldsymbol{c} \leftarrow \boldsymbol{c} + \epsilon \Delta_{\boldsymbol{c}}$
end while

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, and the visible units may be binary or real. There are no intra-layer 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 Fig. 20.1b for an example.

A DBN with L hidden layers contains L weight matrices: $W^{(1)}, \ldots, W^{(L)}$. It

also contains $L+1$ bias vectors: $b^{(0)}, \ldots, b^{(L)}$ with $b^{(0)}$ providing the biases for the visible layer. The probability distribution represented by the DBN is given by

$$p(\boldsymbol{h}^{(L)}, \boldsymbol{h}^{(L-1)}) \propto \exp\left(\boldsymbol{b}^{(L)\top} \boldsymbol{h}^{(L)} + \boldsymbol{b}^{(L-1)\top} \boldsymbol{h}^{(L-1)} + \boldsymbol{h}^{(L-1)\top} \boldsymbol{W}^{(L)} \boldsymbol{h}^{(L)}\right),$$

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

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

In the cause of real-valued visible units, substitute

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

with $\boldsymbol{\beta}$ diagonal for tractability. Generalizations to other exponential family visible units are straightforward, at least in theory. Note that 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.

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 final hidden layers. 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 last two layers.

As a hybrid of directed and undirected models, deep belief networks encounter many of the difficulties associated with both families of models. Because deep belief networks are partially undirected, they require Markov chains for sampling and have an intractable partition function. Because they are directed and generally consist of binary random variables, their evidence lower bound is intractable.

TODO–training procedure TODO–discriminative fine-tuning TODO–view of MLP as variational inference with very loose bound comment on how this does not capture intra-layer explaining away interactions comment on how this does not capture inter-layer feedback interactions TODO–quantitative analysis with AIS TODO–wake sleep?

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 sequential 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, which are Bayesian networks for representing Markov chains.

20.5 Deep Boltzmann Machines

A *deep Boltzmann machine* (DBM) is another kind of deep, generative model (Salakhutdinov and Hinton, 2009a). 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 Fig. 20.2 for the graph structure.

Like RBMs and DBNs, DBMs typically contain only binary units – as we assume in our development of the model – but it may sometimes contain 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:

$$P\left(\boldsymbol{v}, \boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)}, \boldsymbol{h}^{(3)}\right) = \frac{1}{Z(\boldsymbol{\theta})} \exp\left(-E(\boldsymbol{v}, \boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)}, \boldsymbol{h}^{(3)}; \boldsymbol{\theta})\right). \tag{20.20}$$

The DBM energy function is:

$$E(\boldsymbol{v}, \boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)}, \boldsymbol{h}^{(3)}; \boldsymbol{\theta}) = -\boldsymbol{v}^{\top} \boldsymbol{W}^{(1)} \boldsymbol{h}^{(1)} - \boldsymbol{h}^{(1)\top} \boldsymbol{W}^{(2)} \boldsymbol{h}^{(2)} - \boldsymbol{h}^{(2)\top} \boldsymbol{W}^{(3)} \boldsymbol{h}^{(3)}. \tag{20.21}$$

In comparison to the RBM energy function (Eq. 20.9), the DBM energy function includes connections between the hidden units (latent variables) in the form of the weight matrices ($W^{(2)}$ and $W^{(3)}$). 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 similar advantages as offered by

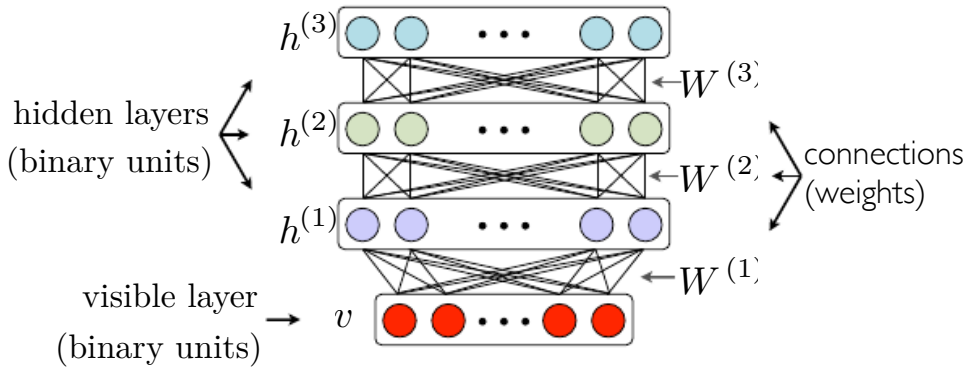

Figure 20.2: The deep Boltzmann machine (offsets on all units are present but suppressed to simplify notation).

the RBM. Specifically, as illustrated in Fig. (TODO: include figure), the DBM layers can be organized into a bipartiite 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.

We show this explicitly for the conditional distribution $P(\boldsymbol{h}^{(1)} = 1 \mid \boldsymbol{v}, \boldsymbol{h}^{(2)})$, in the case of a DBM with two hidden layers (of course, this result generalizes to a DBM with any number of layers).

$$
\begin{aligned}
P(\boldsymbol{h}^{(1)} \mid \boldsymbol{v}, \boldsymbol{h}^{(2)}) &= \frac{P(\boldsymbol{h}^{(1)}, \boldsymbol{v}, \boldsymbol{h}^{(2)})}{P(\boldsymbol{v}, \boldsymbol{h}^{(2)})} \\
&= \frac{\exp\left(\boldsymbol{v}^\top \boldsymbol{W}^{(1)} \boldsymbol{h}^{(1)} + \boldsymbol{h}^{(1)\top} \boldsymbol{W}^{(2)} \boldsymbol{h}^{(2)}\right)}{\sum_{h_1^{(1)}=0}^{1} \cdots \sum_{h_n^{(1)}=0}^{1} \exp\left(\boldsymbol{v}^\top \boldsymbol{W}^{(1)} \boldsymbol{h}^{(1)} + \boldsymbol{h}^{(1)\top} \boldsymbol{W}^{(2)} \boldsymbol{h}^{(2)}\right)} \\
&= \frac{\exp\left(\boldsymbol{v}^\top \boldsymbol{W}^{(1)} \boldsymbol{h}^{(1)} + \boldsymbol{h}^{(1)\top} \boldsymbol{W}^{(2)} \boldsymbol{h}^{(2)}\right)}{\sum_{h_1^{(1)}=0}^{1} \cdots \sum_{h_n^{(1)}=0}^{1} \exp\left(\boldsymbol{v}^\top \boldsymbol{W}^{(1)} \boldsymbol{h}^{(1)} + \boldsymbol{h}^{(1)\top} \boldsymbol{W}^{(2)} \boldsymbol{h}^{(2)}\right)} \\
&= \frac{\exp\left(\sum_{j=1}^{n} \boldsymbol{v}^\top \boldsymbol{W}_{:,j}^{(1)} h_j^{(1)} + h_j^{(1)\top} \boldsymbol{W}_{j,:}^{(2)} \boldsymbol{h}^{(2)}\right)}{\sum_{h_1^{(1)}=0}^{1} \cdots \sum_{h_n^{(1)}=0}^{1} \exp\left(\sum_{j'=1}^{n} \boldsymbol{v}^\top \boldsymbol{W}_{:,j'}^{(1)} h_{j'}^{(1)} + h_{j'}^{(1)\top} \boldsymbol{W}_{j',:}^{(2)} \boldsymbol{h}^{(2)}\right)} \\
&= \frac{\prod_j \exp\left(\boldsymbol{v}^\top \boldsymbol{W}_{:,j}^{(1)} h_j^{(1)} + h_j^{(1)\top} \boldsymbol{W}_{j,:}^{(2)} \boldsymbol{h}^{(2)}\right)}{\sum_{h_1^{(1)}=0}^{1} \cdots \sum_{h_n^{(1)}=0}^{1} \prod_{j'} \exp\left(\boldsymbol{v}^\top \boldsymbol{W}_{:,j'}^{(1)} h_{j'}^{(1)} + h_{j'}^{(1)\top} \boldsymbol{W}_{j',:}^{(2)} \boldsymbol{h}^{(2)}\right)} \\
&= \prod_j \frac{\exp\left(\boldsymbol{v}^\top \boldsymbol{W}_{:,j}^{(1)} h_j^{(1)} + h_j^{(1)\top} \boldsymbol{W}_{j,:}^{(2)} \boldsymbol{h}^{(2)}\right)}{\sum_{h_j^{(1)}=0}^{1} \exp\left(\boldsymbol{v}^\top \boldsymbol{W}_{:,j}^{(1)} h_j^{(1)} + h_j^{(1)\top} \boldsymbol{W}_{j,:}^{(2)} \boldsymbol{h}^{(2)}\right)} \\
&= \prod_j \frac{\exp\left(\boldsymbol{v}^\top \boldsymbol{W}_{:,j}^{(1)} h_j^{(1)} + h_j^{(1)\top} \boldsymbol{W}_{j,:}^{(2)} \boldsymbol{h}^{(2)}\right)}{1 + \exp\left(\boldsymbol{v}^\top \boldsymbol{W}_{:,j}^{(1)} + \boldsymbol{W}_{j,:}^{(2)} \boldsymbol{h}^{(2)}\right)} \\
&= \prod_j P(h_j^{(1)} \mid \boldsymbol{v}, \boldsymbol{h}^{(2)}). \qquad (20.22)
\end{aligned}
$$

From the above we can conclude that the conditional distribution for any layer of the DBM conditioned on the neighboring layers, is fractorial (i.e. all variables in the layer are conditionally independent). Further, we've shown that

this conditional distribution is given by a logistic sigmoid function:

$$\begin{aligned} P(h_j^{(1)} = 1 \mid \boldsymbol{v}, \boldsymbol{h}^{(2)}) &= \frac{\exp\left(\boldsymbol{v}^\top \boldsymbol{W}_{:,j}^{(1)} + \boldsymbol{W}_{j,:}^{(2)} \boldsymbol{h}^{(2)}\right)}{1 + \exp\left(\boldsymbol{v}^\top \boldsymbol{W}_{:,j}^{(1)} + \boldsymbol{W}_{j,:}^{(2)} \boldsymbol{h}^{(2)}\right)} \\ &= \frac{1}{1 + \exp\left(-\boldsymbol{v}^\top \boldsymbol{W}_{:,j}^{(1)} - \boldsymbol{W}_{j,:}^{(2)} \boldsymbol{h}^{(2)}\right)} \\ &= \text{sigmoid}\left(\boldsymbol{v}^\top \boldsymbol{W}_{:,j}^{(1)} + \boldsymbol{W}_{j,:}^{(2)} \boldsymbol{h}^{(2)}\right). \end{aligned} \tag{20.23}$$

For the two layer DBM, the conditional distributions of the remaining two layers $(\boldsymbol{v}, \boldsymbol{h}^{(2)})$ also factorize. That is $P(\boldsymbol{v} \mid \boldsymbol{h}^{(1)}) = \prod_{i=1}^{d} P(v_i \mid h^{(1)})$, where

$$P(v_i = 1 \mid h^{(1)}) = \text{sigmoid}\left(\boldsymbol{W}_{i,:}^{(1)} h^{(1)}\right). \tag{20.24}$$

Also, $P(h^{(2)} \mid h^{(1)}) = \prod_{k=1}^{m} P(h_k^{(2)} \mid h^{(1)})$, where

$$P(h_k^{(2)} = 1 \mid h^{(1)}) = \text{sigmoid}\left(h^{(1)\top} \boldsymbol{W}_{:,k}^{(2)}\right). \tag{20.25}$$

20.5.1 Interesting Properties

TODO: comparison to DBNs TODO: comparison to neuroscience (local learning) "most biologically plausible" TODO: description of easy mean field TODO: description of sampling, comparison to general Boltzmann machines,DBNs

20.5.2 DBM Mean Field Inference

For the two hidden layer DBM, the conditional distributions, $P(\boldsymbol{v} \mid \boldsymbol{h}^{(1)})$, $P(\boldsymbol{h}^{(1)} \mid \boldsymbol{v}, \boldsymbol{h}^{(2)})$, and $P(\boldsymbol{h}^{(2)} \mid \boldsymbol{h}^{(1)})$ are factorial, however the posterior distribution over all the hidden units given the visible unit, i.e. $P(\boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)} \mid \boldsymbol{v})$, can be complicated. This is, of course, due to the interaction weights $\boldsymbol{W}^{(2)}$ between $\boldsymbol{h}^{(1)}$ and $\boldsymbol{h}^{(2)}$ which render these variables mutually dependent, given an observed $\boldsymbol{v}$.

So, like 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 Sec. 19.1), specifically a mean field approximation. The mean field approximation is a simple form of variational inference, where we restrict the approximating distribution to fully factorial distributions. In the context of DBMs, the mean field equations capture the bidirectional interactions between layers. In this section we derive the iterative approximate inference procedure originally introduced in Salakhutdinov and Hinton (2009a)

In variational approximations to inference, we approach the task of approximating a particular target distribution – in our case, the posterior distribution over the hidden units given the visible units – by some reasonably simple family of distributions. In the case of the mean field approximation, the approximating family is the set of distributions where the hidden units are conditionally independent.

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

$$Q(\boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)} \mid \boldsymbol{v}) = \prod_{j=1}^{n} Q(h_j^{(1)} \mid \boldsymbol{v}) \prod_{k=1}^{m} Q(h_k^{(2)} \mid \boldsymbol{v}). \tag{20.26}$$

The mean field approximation attempts to find *for every observation* a member of this family of distributions that "best fits" the true posterior $P(\boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)} \mid \boldsymbol{v})$. By best fit, we specifically mean that we wish to find the approximation Q that minimizes the KL-divergence with P, i.e. $\mathrm{KL}(Q \| P)$ where:

$$\mathrm{KL}(Q \| P) = \sum_{\boldsymbol{h}} Q(\boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)} \mid \boldsymbol{v}) \log \left(\frac{Q(\boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)} \mid \boldsymbol{v})}{P(\boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)} \mid \boldsymbol{v})} \right) \tag{20.27}$$

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 considering here) there is no loss of generality by fixing a parametrization of the model in advance.

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

$$\begin{aligned} Q(\boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)} \mid \boldsymbol{v}) &= \prod_{j=1}^{n} Q(h_j^{(1)} \mid \boldsymbol{v}) \prod_{k=1}^{m} Q(h_k^{(2)} \mid \boldsymbol{v}) \\ &= \prod_{j=1}^{n} (\hat{h}_j^{(1)})^{h_j^{(1)}} (1 - \hat{h}_j^{(1)})^{(1 - h_j^{(1)})} \times \prod_{k=1}^{m} (\hat{h}_k^{(2)})^{h_k^{(2)}} (1 - \hat{h}_k^{(2)})^{(1 - h_k^{(2)})} \end{aligned} \tag{20.28}$$

Of course, for DBMs with more layers the approximate posterior parametrization can be extended in the obvious way.

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. One way to do this is to explicitly minimize $\mathrm{KL}(Q\|P)$ with respect to the variational parameters of Q. We will approach the selection of Q from a slightly different, but entirely equivalent, path. Rather than minimize $\mathrm{KL}(Q\|P)$, we will maximize the variational lower bound (or evidence lower bound: see Sec. 19.1), which in the context of the 2-hidden-layer deep Boltzmann machine is given by:

$$\begin{aligned}\mathcal{L}(Q) &= \sum_{\boldsymbol{h}^{(1)},\boldsymbol{h}^{(2)}} Q(\boldsymbol{h}^{(1)},\boldsymbol{h}^{(2)} \mid \boldsymbol{v}) \log \left(\frac{P(\boldsymbol{v},\boldsymbol{h}^{(1)},\boldsymbol{h}^{(2)};\boldsymbol{\theta})}{q(\boldsymbol{h}^{(1)},\boldsymbol{h}^{(2)} \mid \boldsymbol{v})} \right) \\ &= -\sum_{\boldsymbol{h}^{(1)},\boldsymbol{h}^{(2)}} Q(\boldsymbol{h}^{(1)},\boldsymbol{h}^{(2)} \mid \boldsymbol{v}) E(\boldsymbol{v},\boldsymbol{h}^{(1)},\boldsymbol{h}^{(2)};\boldsymbol{\theta}) - \log Z(\boldsymbol{\theta}) + \mathcal{H}(Q), \quad (20.29)\end{aligned}$$

where $Z(\boldsymbol{\theta})$ is the DBM partition function and $\mathcal{H}(Q)$ is the entropy of the mean field distribution.

We wish to maximize the variational lower bound in Eq. 20.29 with respect to the mean field parameters of $Q(\boldsymbol{h}^{(1)},\boldsymbol{h}^{(2)} \mid \boldsymbol{v})$. Substituting Eq. 20.28 for $Q(\boldsymbol{h}^{(1)},\boldsymbol{h}^{(2)} \mid \boldsymbol{v})$ in the variational lower bound, we get:

$$\mathcal{L}(q) = \sum_i \sum_{j'} v_i W^{(1)}_{ij'} \hat{h}^{(1)}_{j'} + \sum_{j'} \sum_{k'} \hat{h}^{(1)}_{j'} W^{(2)}_{j'k'} \hat{h}^{(2)}_{k'} - \ln Z(\theta) + \mathcal{H}(q). \quad (20.30)$$

We maximize the above expression (Eq. 20.30) by taking deriatives with respect to the variational parameters and solving for the system of fixed point equations:

$$\frac{\partial}{\partial \hat{h}^{(1)}_j} \mathcal{L}(q) = 0 \quad \forall j \in \{1,\ldots,n\}, \qquad \frac{\partial}{\partial \hat{h}^{(2)}_k} \mathcal{L}(q) = 0 \quad \forall k \in \{1,\ldots,m\}$$

The gradient with respect to, for example, $\hat{h}^{(1)}_j$ is reasonable straightforward

to evaluate:

$$\begin{aligned}
\frac{\partial}{\partial \hat{h}_j^{(1)}}\mathcal{L}(q) &= \frac{\partial}{\partial \hat{h}_j^{(1)}}\left[\sum_i \sum_{j'} v_i W_{ij'}^{(1)} \hat{h}_{j'}^{(1)} + \sum_{j'}\sum_{k'} \hat{h}_{j'}^{(1)} W_{j'k'}^{(2)} \hat{h}_{k'}^{(2)} - \ln Z(\theta) + \mathcal{H}(q)\right] \\
&= \frac{\partial}{\partial \hat{h}_j^{(1)}}\left[\sum_i \sum_{j'} v_i W_{ij'}^{(1)} \hat{h}_{j'}^{(1)} + \sum_{j'}\sum_{k'} \hat{h}_{j'}^{(1)} W_{j'k'}^{(2)} \hat{h}_{k'}^{(2)} - \ln Z(\theta)\right. \\
&\qquad - \sum_{j'}\Big(\hat{h}_{j'}^{(1)} \ln \hat{h}_{j'}^{(1)} + (1-\hat{h}_{j'}^{(1)})\ln(1-\hat{h}_{j'}^{(1)})\Big) \\
&\qquad \left. - \sum_{k'}\Big(\hat{h}_{k'}^{(2)} \ln \hat{h}_{k'}^{(2)} + (1-\hat{h}_{k'}^{(2)})\ln(1-\hat{h}_{k'}^{(2)})\Big)\right] \\
&= \sum_i v_i W_{ij}^{(1)} + \sum_{k'} W_{jk'}^{(2)} \hat{h}_{k'}^{(2)} - \ln\left(\frac{\hat{h}_{j'}^{(1)}}{1-\hat{h}_{j'}^{(1)}}\right),
\end{aligned}$$

where in the second line, we have just expanded the terms involved in the entropy $\mathcal{H}(q)$. Setting this derivative to zero and solving for $\hat{h}_j^{(1)}$, we have

$$\begin{aligned}
\frac{\partial}{\partial \hat{h}_j^{(1)}}\mathcal{L}(q) = 0 = \sum_i v_i W_{ij}^{(1)} + \sum_{k'} W_{jk'}^{(2)} \hat{h}_{k'}^{(2)} - \ln \frac{\hat{h}_j^{(1)}}{1-\hat{h}_j^{(1)}}\Bigg) \\
\hat{h}_j^{(1)} = \text{sigmoid}\left(\sum_i v_i W_{ij}^{(1)} + \sum_{k'} W_{jk'}^{(2)} \hat{h}_{k'}^{(2)}\right)
\end{aligned}$$

A similar derivation leads to the other set of equations for the second hidden layer variational parameters. Putting these together, we have the following system of equations:

$$\hat{h}_j^{(1)} = \text{sigmoid}\left(\sum_i v_i W_{ij}^{(1)} + \sum_{k'} W_{jk'}^{(2)} \hat{h}_{k'}^{(2)}\right), \quad \forall j \tag{20.31}$$

$$\hat{h}_k^{(2)} = \text{sigmoid}\left(\sum_{j'} W_{j'k}^{(2)} \hat{h}_{j'}^{(1)}\right), \quad \forall k \tag{20.32}$$

At a fixed point of this system of equations, we have a local maximum of our variational lower bound $\mathcal{L}(q)$. Thus they define a iterative algorithm where we intersperse updates of $\hat{h}_j^{(1)}$ (using Eq. 20.31) and updates of $\hat{h}_k^{(2)}$ (using Eq. 20.32). So variational inference in the two hidden layer deep Boltzmann machine amounts to iterating these update equations for $\hat{h}_j^{(1)}$ and $\hat{h}_k^{(2)}$ until convergence. In practice, ≈ 10 iterations is usually sufficient. Extending approximate variational inference to deeper DBMs is straightforward.

20.5.3 DBM Parameter Learning

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.

The posterior distribution over the hidden units in a deep Boltzmann machine is intractable, due to the interactions between different hidden layers. This means that we must use approximate inference during learning. The standard approach is to use stochastic gradient ascent on the mean field lower bound, as described in chapter 19. Mean field is incompatible with most of the methods for approximating the gradients of the log partition function described in chapter 18. Moreover, it has been observed that for contrastive divergence to work well, it is important that the samples from the posterior (e.g., for the 2 hidden layer DBM: $P(\boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)} \mid \boldsymbol{v})$) be exact (Salakhutdinov and Hinton, 2009b). In the case of the DBM, the intractability of the posterior means that we would have to run a Gibbs sampler until the samples converged to samples from the true posterior (i.e. until they "burned in"). Thus for the DBM, CD offers no speedup relative to naive MCMC methods. Instead, DBMs are usually trained using a variant of stochastic maximum likelihood. The negative phase samples can be generated simply by running a Gibbs sampling chain that alternates between sampling the odd-numbered layers and sampling the even-numbered layers.

Learning in the DBM can equivalently be considered as performing a variational form of the Expectation Maximization (EM) algorithm. Specifically, consider the variational lower bound for the two-layer DBM (making the dependency on the model parameters explicit):

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

This expression lower bounds the likelihood $P(\boldsymbol{v} \mid \boldsymbol{\theta})$. So by maximizing this bound we hope to improve the likelihood. Thus we can think of $\mathcal{L}(Q, \boldsymbol{\theta})$ as a surrogate objective function for the DBM. From this perspective it is natural to consider a 2-step optimization procedure. In the first step (the E-step or expectation step), we optimize $\mathcal{L}(Q, \boldsymbol{\theta})$ with respect to the variational parameters. In the case of the two-layer DBM this amounts to solving for $\hat{\boldsymbol{h}}^{(1)}$ and $\hat{\boldsymbol{h}}^{(2)}$ via the iterative scheme introduced above. Then in the second step (the M-step or maximization step), we optimize $\mathcal{L}(Q, \boldsymbol{\theta})$ with respect to the model parameters $\boldsymbol{\theta}$.

Note that maximizing the variational lower bound with respect to the parameters does not guarantee that we improve the true likelihood $P(\boldsymbol{v} \mid \boldsymbol{\theta})$ on every step. [3] That said, in practice we often find that we are able to make progress in training DBMs by maximizing the lower bound $\mathcal{L}(Q, \boldsymbol{\theta})$.

Unlike the standard M-step we typically have as part of the EM algorithm, our M-step will not actually maximize $\mathcal{L}(Q, \boldsymbol{\theta})$ with respect to $\boldsymbol{\theta}$ (holding Q fixed). The presence of the partition function makes it impractical to solve the system of equations $\nabla_{\boldsymbol{\theta}} \mathcal{L}(Q, \boldsymbol{\theta}) = \mathbf{0}$ for $\boldsymbol{\theta}$. Instead we will be content to make incremental progress toward this maximum by taking a small step in the direction of the gradient $\nabla_{\boldsymbol{\theta}} \mathcal{L}(Q, \boldsymbol{\theta})$. In the case of the 2-hidden layer DBM, this is given by:

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}} \mathcal{L}(Q, \boldsymbol{\theta}) &= \frac{\partial}{\partial \theta} \left(\sum_i \sum_{j'} v_i W^{(1)}_{ij'} \hat{h}^{(1)}_{j'} + \sum_{j'} \sum_{k'} \hat{h}^{(1)}_{j'} W^{(2)}_{j'k'} \hat{h}^{(2)}_{k'} - \ln Z(\theta) + \mathcal{H}(Q) \right) \\
&= \frac{\partial}{\partial \theta} \left(\sum_i \sum_{j'} v_i W^{(1)}_{ij'} \hat{h}^{(1)}_{j'} + \sum_{j'} \sum_{k'} \hat{h}^{(1)}_{j'} W^{(2)}_{j'k'} \hat{h}^{(2)}_{k'} \right) - \frac{\partial}{\partial \theta} \ln Z(\theta)
\end{aligned} \tag{20.33}$$

The first term in Eq. 20.33 is straightforward, once the values of $\hat{\boldsymbol{h}}^{(1)}$ and $\hat{\boldsymbol{h}}^{(2)}$ have been computed in the E-step. Our use of variation approximate inference has rendered learning in the DBM as analogous to training in the RBM where the likelihood gradient (Eq. 20.14) is also composed of a analytically tractable term and a term involving the gradient of the partition function:

$$- n \mathbb{E}_{P(\boldsymbol{v}, \boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)})} \left[\nabla_{\boldsymbol{\theta}} - E(\boldsymbol{v}, \boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)}) \right] \tag{20.34}$$

Similar to the RBM case, the partition function's contribution to the gradient of the variational lower bound is intractable. We approximate it using a variational version of stochastic maximum likelihood [4] (VSML) algorithm. The non-variational version of stochastic maximum likelihood algorithm is discussed in Sec. 18.2 and is applied to RBMs in Sec. 20.3.2.

Unlike in the RBM, the interaction between the hidden units of the DBM precludes a direct application of the contrastive divergence training algorithm. Specifically the issue is that, in the positive phase, in order to get samples from

[3] In standard EM we do have just a guarantee. The difference is that in the case of standard EM we assume the true posterior is tractable and therefore we can set $Q(\boldsymbol{h} \mid \boldsymbol{v}, \boldsymbol{\theta}^{(t)}) = P(\boldsymbol{h} \mid \boldsymbol{v}, \boldsymbol{\theta}^{(t)})$. Under these conditions the lower bound is tight, i.e. $\mathcal{L}(Q, \boldsymbol{\theta}) = P(\boldsymbol{v} \mid \boldsymbol{\theta})$

[4] Salakhutdinov and Hinton (2009a) refer to this algorithm as persistent contrastive divergence. We prefer to distinguish the variational version of the algorithm as applied to DBMs from the original stochastic maximum likelihood algorithm that directly (though stochastically) maximizes the likelihood rather than a lower bound on the likelihood as we are doing here.

the posterior $P(\boldsymbol{h} \mid \boldsymbol{v})$, one may have to wait a significant amount of time for the samples to "burn-in". The necessity for this burn-in renders CD an impractical algorithm for training CD. As far as we known, variants of CD that make use of the variational approximation for the positive phase gradient approximation have not been unexplored.

Variational stochastic maximum likelihood as applied to the DBM is given in Algorithm 20.3. Recall that we have included the offset parameters in the weight matrices $\boldsymbol{W}^{(1)}$ and $\boldsymbol{W}^{(2)}$. Note that the Gibbs sampling in the negative phase of the stochastic maximum likelihood algorithm can be divided into two blocks of updates, one including all odd layers (including the visible layer) and the other including all even layers. Due to the 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.

20.5.4 Practical Training Strategies

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. Note that a DBM with very small weights in all but the first layer represents approximately the same distribution as an RBM.

It is not clear exactly why this happens. When DBMs are initialized from a pretrained configuration, training usually succeeds. See section 20.5.4 for details. One possibility is that it is difficult to coordinate the learning rate of the stochastic gradient algorithm with the number of Gibbs steps used in the negative phase of stochastic maximum likelihood. SML relies on the learning rate being small enough relative to the number of Gibbs steps that the Gibbs chain can mix again after each update to the model parameters. The distribution represented by the model can change very rapidly during the earlier parts of training, and this may make it difficult for the negative chains employed by SML to fully mix. As described in section 20.5.5, *multi-prediction deep Boltzmann machines* avoid the potential inaccuracy of SML by training with a different objective function that is less principled but easier to compute. Another possible explanation for the failure of joint training with mean field and SML is that the Hessian matrix could be poorly conditioned. This perspective motivates *centered deep Boltzmann machines*, presented in section 20.5.6, which modify the model family in order to obtain a better conditioned Hessian matrix.

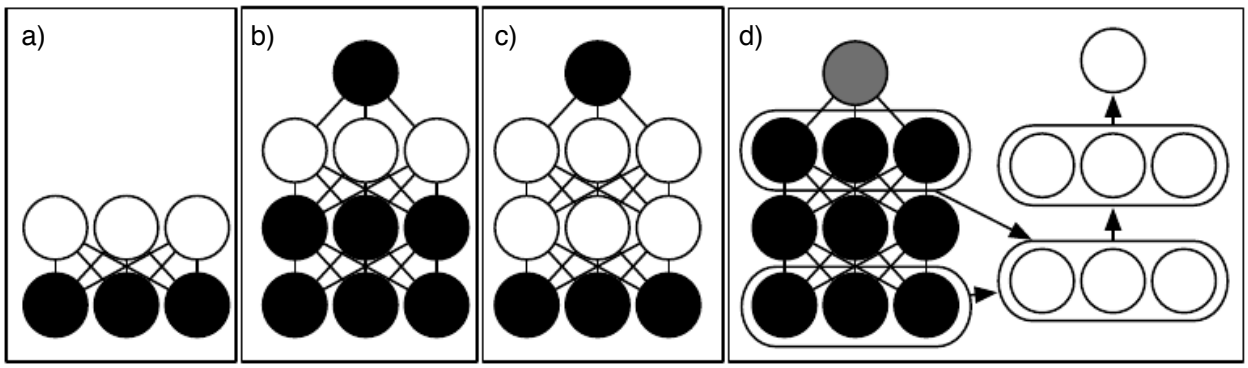

Figure 20.3: The deep Boltzmann machine training procedure used to obtain the state of the art classification accuracy on the MNIST dataset (Srivastava *et al.*, 2014; Salakhutdinov and Hinton, 2009a). TODO: this is not state of the art anymore, just best DBM result a) Train an RBM by using CD to approximately maximize $logP(\boldsymbol{v})$. b) Train a second RBM that models $\boldsymbol{h}^{(1)}$ and 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 $logP(\mathbf{v}, \mathrm{y})$ 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).

Layerwise Pretraining

The original and most popular method for overcoming the joint training problem of DBMs is greedy layerwise 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 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 only make 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.

Note that this greedy layerwise 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. However, in the case of the greedy layerwise training procedure, we actually use a different objective function at each step.

TODO: details of combining stacked RBMs into a DBM TODO: partial mean field negative phase

20.5.5 Multi-Prediction Deep Boltzmann Machines

TODO– cite stoyanov TODO

20.5.6 Centered Deep Boltzmann Machines

TODO

This chapter has described the tools needed to fit a very broad class of probabilistic models. Which tool to use depends on which aspects of the log-likelihood are problematic.

For the simplest distributions p, the log-likelihood is tractable, and the model can be fit with a straightforward application of maximum likelihood estimation and gradient ascent as described in chapter

In this chapter, I've shown what to do in two different difficult cases. If Z is intractable, then one may still use maximum likelihood estimation via the sampling approximation techniques described in section 18.2. If $p(h \mid v)$ is intractable, one may still train the model using the negative variational free energy rather than the likelihood, as described in 19.6.

It is also possible that *both* of these difficulties will arise. An example of this occurs with the *deep Boltzmann machine* (Salakhutdinov and Hinton, 2009b), which is essential a sequence of RBMs composed together. The model is depicted graphically in Fig. 20.1c.

This model still has the same problem with computing the partition function as the simpler RBM does. It has also discarded the restricted structure that made $P(h \mid v)$ easy to represent in the RBM. The typical way to train the DBM is to minimize the variational free energy rather than maximize the likelihood. Of course, the variational free energy still depends on the partition function, so it is necessary to use sampling techniques to approximate its gradient.

TODO: k-NADE

20.6 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 (TODO cite some examples). However, this is not a particularly theoretically satisfying approach.

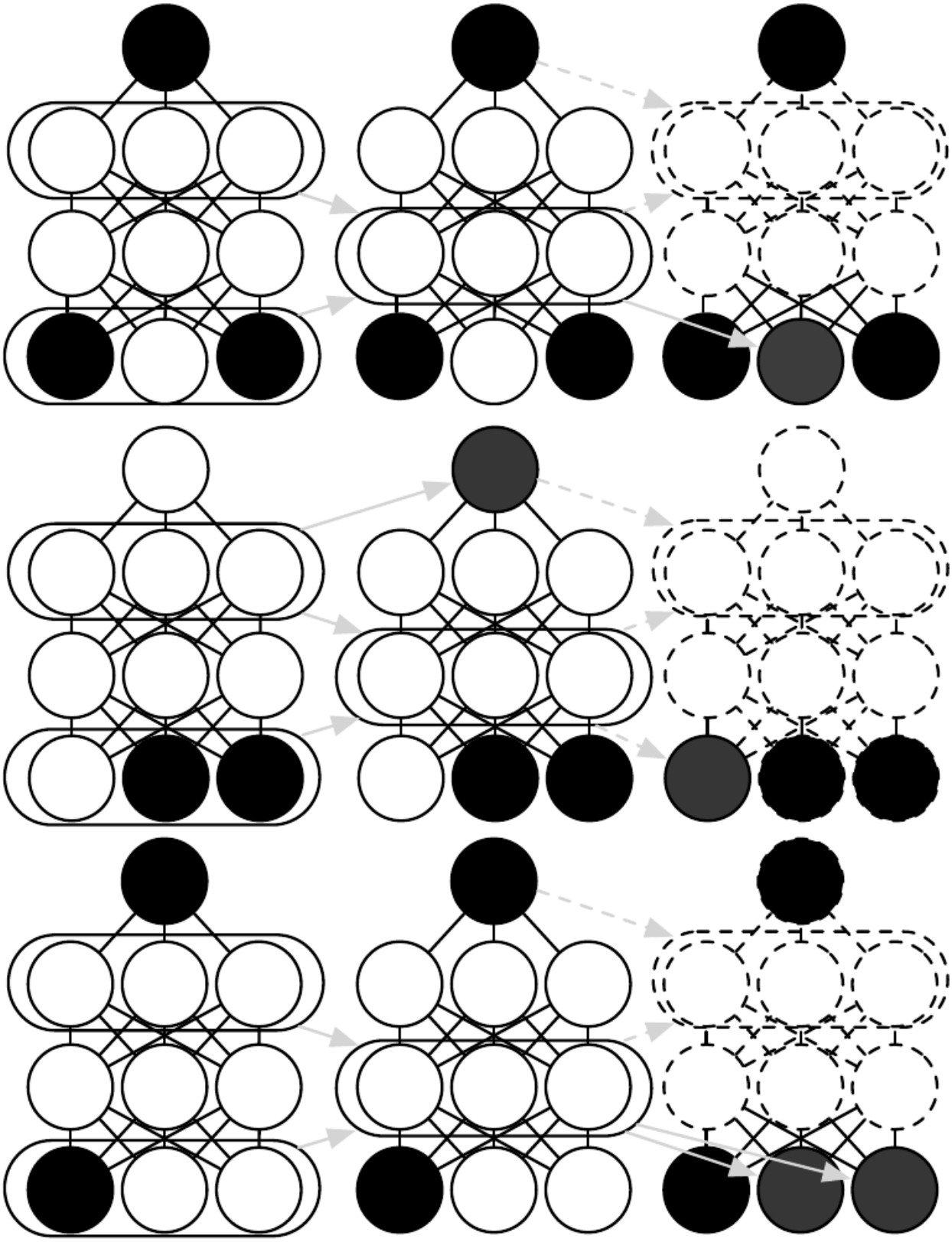

Figure 20.4: TODO caption and label, reference from text Figure reprinted from (Goodfellow *et al.*, 2013b).

20.6.1 Gaussian-Bernoulli RBMs

TODO– cite exponential family harmoniums? TODO– multiple ways of parametrizing them (citations?)

20.6.2 mcRBMs

TODO–mcRBMs [5] TODO–HMC

20.6.3 mPoT Model

TODO–mPoT

20.6.4 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.

The spike and slab RBM has two sets of hidden units: the *spike* units $\mathbf{h}$ which are binary, and the slab units $\mathbf{s}$ which are real-valued. 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 be appear in the input. The corresponding spike variable h_i determines whether that component is present at all. The corresponding slab variable s_i determines the brightness 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.

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.

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

[5] 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."

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 Fig. 13.1.

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.*, 2013c), also known as S3C.

20.7 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 $p = \max_i \mathrm{d}_i$ 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 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 pool 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 [6]. This model is able to perform operations such as filling in missing portions of its input. However, it has not proven especially useful as a pretraining strategy for supervised learning, performing similarly to shallow baseline models introduced by Pinto *et al.* (2008).

TODO: comment on partition function changing when you change the image size, boundary issues

20.8 Other Boltzmann Machines

TODO–Conditional Boltzmann machine TODO-RNN-RBM TODO–discriminative Boltzmann machine TODO–Heng's class relevant and irrelevant Boltzmann machines TODO– Honglak's recent work

20.9 Directed Generative Nets

So far in this chapter we have focused on undirected generative models, in the deep learning context these are almost always parametrized via an energy function E and possess an intractable partition function Z. The exception being the deep belief net which can be characterized as a hybrid directed / undirected model.

As discussed in Chapter 13, directed graphical models make up a second prominent class of graphical models. While directed graphical models have been the very popular within the greater Machine Learning community, within the smaller Deep Learning community they have until recently been overshadowed by undirected models such as the RBM.

In this section we will consider some of the standard directed graphical models that have traditionally associated with the deep learning community [7].

[6] 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.

[7] The list of directed graphical models that we cover here is inevitably going to be incomplete. The choice of models we include has more to do their prominence within the Deep Learning

TODO: sigmoid belief nets –¿ for fully observed sigmoid nets, coordinate with sec:autoregressive-nets which comes next

TODO: refer back to DBN TODO: sparse coding (maybe drop this from the list, covered elsewhere) TODO: deconvolutional nets? (AC votes for dropping this) TODO: refer back to S3C and BSC (binary sparse coding) TODO: NADE will be in RNN chapter, refer back to it here make sure k-NADE and multi-NADE are mentioned somewhere

TODO: refer to DARN and NVIL?

TODO: Stochastic Feedforward nets

20.9.1 Sigmoid Belief Nets

Sigmoid Belief Nets were originally conceived in response to the Neal (1992)

is one of the first

20.9.2 Differentiable Generator Nets

TODO describe how VAEs and GANs both use the same kind of generator net cite the generating chairs paper to show how this generator net can be trained with a procedure that isn't explicitly unsupervised cite both Kevin and Zoubin's version of training a generator net with MMD

20.9.3 Variational Autoencoders

The variational autoencoder is model

$$\mathcal{L} \tag{20.35}$$

TODO

20.9.4 Variational Interpretation of PSD

TODO, develop the explanation of Sec. 9.1 of Bengio *et al.* (2013c).

20.9.5 Generative Adversarial Networks

TODO: do we want to still use the capital value function here? Should we say it's a functional? Note that the G is OK because it's a distribution

Generative adversarial networks (TODO cite) are another kind of generative model based on differentiable mappings from input noise to samples that resemble the data. In this sense, they closely resemble variational autoencoders. However,

community and the perceived impact that they have had on the community.

the training procedure is different, and generative model is not necessarily coupled with an inference network. It is theoretically possible to train an inference network using a strategy similar to the wake-sleep algorithm, but there is no need to infer posterior variables during training.

Generative adversarial networks are based on game theory. A *generator network* is trained to map input noise $\boldsymbol{z}$ to samples $\boldsymbol{x}$. This function $g(\boldsymbol{z})$ defines the generative model. The distribution $p(\boldsymbol{z})$ is not learned; it is simply fixed to some distribution at the start of training (usually a very unstructured distribution such as a normal or uniform distribution). We can think of $g(\boldsymbol{z})$ as defining a conditional distribution

$$p(\boldsymbol{x} \mid \boldsymbol{z}) = \mathcal{N}(\boldsymbol{x} \mid g\left(\boldsymbol{z}\right), \frac{1}{\beta}\boldsymbol{I}\right),$$

but in all learning rules we take limit as $\beta \to \infty$ so we can treat $g(\boldsymbol{z})$ itself as a sample and ignore the parametrization of the output distribution.

The generator g is pitted against an adversary: a discriminator network d. The discriminator network receives data or samples $\boldsymbol{x}$ as input and outputs its estimate of the probability that $\boldsymbol{x}$ was sampled from the data rather than the model. During training, d tries to maximize and g tries to minimize a value function measuring the log probability of d being correct:

$$g^* = \arg\min_g \max_d V(g, d)$$

where

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

The optimization of g can be done simply by backpropagating through d then g, so the learning process requires neither approximate inference nor approximation of a partition function gradient. In the case where $\max_d v(g, d)$ is convex (such as the case where optimization is performed directly in the space of probability density functions) then the procedure is guaranteed to converge and is asymptotically consistent. In practice, the procedure can be difficult to make work, because it can be difficult to keep d optimized well enough to provide a good estimate of how to update g at all times.

20.9.6 Convolutional Generative Networks

TODO– discuss convolutional generator nets (was GANs paper the first?) and be sure to cover "unpooling" include the unpooling technique from generatoring chairs paper, and include others if there are relevant others

20.10 Auto-Regressive Networks

Auto-regressive networks are similar to recurrent networks in the sense that we also decompose a joint probability over the observed variables as a product of conditionals of the form $P(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \ldots, \boldsymbol{x}_1)$ but we drop the form of parameter sharing that makes these conditionals all share the same parametrization across time. This makes sense when the variables are *not* elements of a translation-equivariant sequence (see Section 9.2 for more on equivariance), but instead form an arbitrary tuple without any particular ordering that would correspond to a translation-equivariant form of relationship between variables at position k and variables at position k'. Such models have been called *fully-visible Bayes networks* (Frey *et al.*, 1996) and used successfully in many forms, first with logistic regression for each conditional distribution (Frey, 1998) and then with neural networks (Bengio and Bengio, 2000b; Larochelle and Murray, 2011). In some forms of auto-regressive networks, such as NADE (Larochelle and Murray, 2011), described in Section 20.10.3 below, we can re-introduce a form of parameter sharing that is different from the one found in recurrent networks, but that brings both a statistical advantage (less parameters) and a computational advantage (less computation). Although we drop the sharing over time, as we see below in Section 20.10.2, using a deep learning concept of *reuse of features*, we can *share* features that have been computed for predicting $\boldsymbol{x}_{t-k}$ with the sub-network that predicts $\boldsymbol{x}_t$.

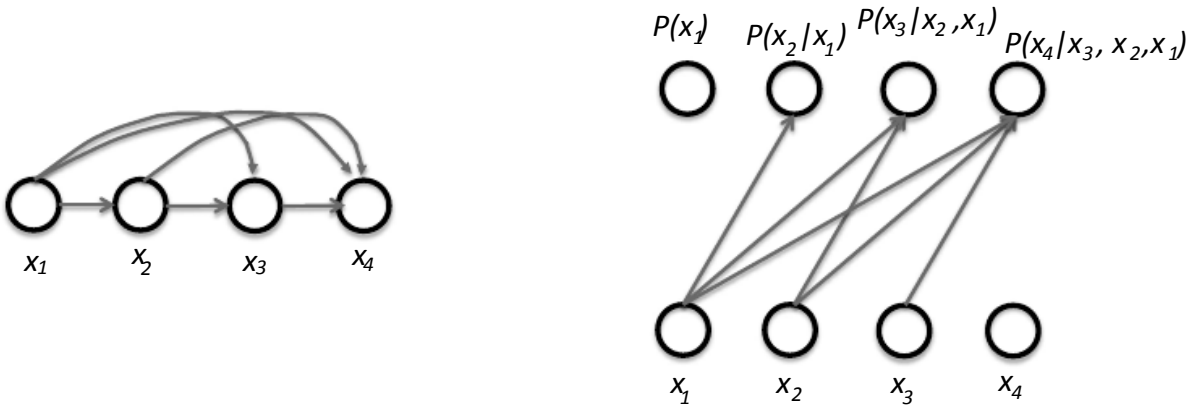

Figure 20.5: An auto-regressive network predicts the i-th variable from the $i-1$ previous ones. Left: corresponding graphical model (which is the same as that of a recurrent network). Right: corresponding computational graph, in the case of the logistic auto-regressive network, where each prediction has the form of a logistic regression, i.e., with i free parameters (for the $i-1$ weights associated with $i-1$ inputs, and an offset parameter).

20.10.1 Logistic Auto-Regressive Networks

Let us first consider the simplest auto-regressive network, without hidden units, and hence no sharing at all. Each $P(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \ldots, \boldsymbol{x}_1)$ is parametrized as a linear model, e.g., a logistic regression. This model was introduced by Frey (1998) and has $O(T^2)$ parameters when there are T variables to be modeled. It is illustrated in Fig. 20.5, showing both the graphical model (left) and the computational graph (right).

A clear disadvantage of the logistic auto-regressive network is that one cannot easily increase its capacity in order to capture more complex data distributions. It defines a parametric family of fixed capacity, like the linear regression, the logistic regression, or the Gaussian distribution. In fact, if the variables are continuous, one gets a linear auto-regressive model, which is thus another way to formulate a Gaussian distribution, i.e., only capturing pairwise interactions between the observed variables.

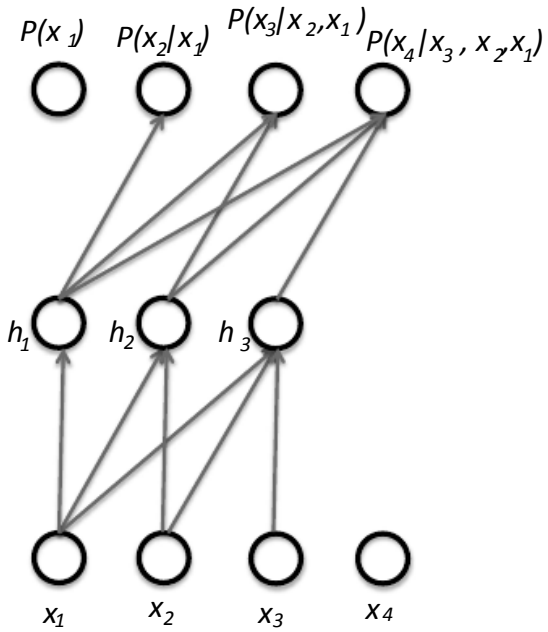

Figure 20.6: A neural auto-regressive network predicts the i-th variable $\boldsymbol{x}_i$ from the $i-1$ previous ones, but is parametrized so that features (groups of hidden units denoted h_i) that are functions of $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_i$ can be reused in predicting all of the subsequent variables $\boldsymbol{x}_{i+1}, \boldsymbol{x}_{i+2}, \ldots$.

20.10.2 Neural Auto-Regressive Networks

Neural Auto-Regressive Networks have the same left-to-right graphical model as logistic auto-regressive networks (Fig. 20.5, left) but a different parametrization that is at once more powerful (allowing to extend the capacity as needed and

approximate any joint distribution) and can improve generalization by introducing a parameter sharing and feature sharing principle common to deep learning in general. The first paper on neural auto-regressive networks by Bengio and Bengio (2000b) (see also Bengio and Bengio (2000a) for the more extensive journal version) were motivated by the objective to avoid the curse of dimensionality arising out of traditional non-parametric graphical models, sharing the same structure as Fig. 20.5 (left). In the non-parametric discrete distribution models, each conditional distribution is represented by a table of probabilities, with one entry and one parameter for each possible configuration of the variables involved. By using a neural network instead, two advantages are obtained:

1. The parametrization of each $P(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \ldots, \boldsymbol{x}_1)$ by a neural network with $(t-1) \times k$ inputs and k outputs (if the variables are discrete and take k values, encoded one-hot) allows one to estimate the conditional probability without requiring an exponential number of parameters (and examples), yet still allowing to capture high-order dependencies between the random variables.

2. Instead of having a different neural network for the prediction of each $\boldsymbol{x}_t$, a *left-to-right* connectivity illustrated in Fig. 20.6 allows one to merge all the neural networks into one. Equivalently, it means that the hidden layer features computed for predicting $\boldsymbol{x}_t$ can be reused for predicting $\boldsymbol{x}_{t+k}$ $(k > 0)$. The hidden units are thus organized in *groups* that have the particularity that all the units in the t-th group only depend on the input values $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_t$. In fact the parameters used to compute these hidden units are jointly optimized to help the prediction of all the variables $\boldsymbol{x}_{t+1}, \boldsymbol{x}_{t+2}, \ldots$. This is an instance of the *reuse principle* that makes *multi-task learning* and *transfer learning* successful with neural networks and deep learning in general (See Sections 7.12 and 16.2).

Each $P(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \ldots, \boldsymbol{x}_1)$ can represent a conditional distribution by having outputs of the neural network predict *parameters* of the conditional distribution of $\boldsymbol{x}_t$, as discussed in Section 6.3.2. Although the original neural auto-regressive networks were initially evaluated in the context of purely discrete multivariate data (e.g., with a sigmoid output - Bernoulli case - or softmax output - multinoulli case) it is natural to extend such models to continuous variables or joint distributions involving both discrete and continuous variables. For example, Uria *et al.* (2013) developed an extension to real-valued variables called RNADE.

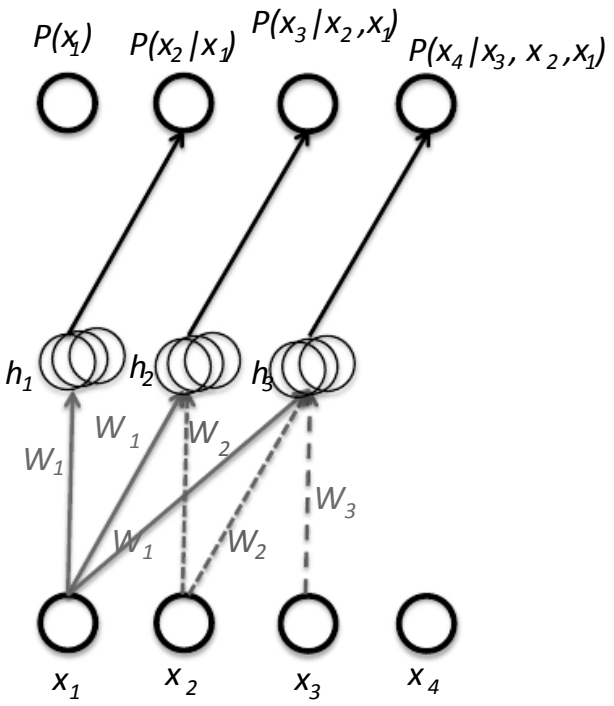

Figure 20.7: NADE (Neural Auto-regressive Density Estimator) is a neural auto-regressive network, i.e., the hidden units are organized in groups $\boldsymbol{h}_j$ so that only the inputs $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_i$ participate in computing $\boldsymbol{h}_i$ and predicting $P(\boldsymbol{x}_j \mid \boldsymbol{x}_{j-1}, \ldots, \boldsymbol{x}_1)$, for $j > i$. The particularity of NADE is the use of a particular weight sharing pattern: the same $W'_{jki} = W_{ki}$ is shared (same color and line pattern in the figure) for all the weights outgoing from $\boldsymbol{x}_i$ to the k-th unit of any group $j \geq i$. The vector $(W_{1i}, W_{2i}, \ldots)$ is denoted $\boldsymbol{W}_{:,i}$ here.

20.10.3 NADE

A very successful recent form of neural auto-regressive network was proposed by Larochelle and Murray (2011). The architecture is basically the same as for the original neural auto-regressive network of Bengio and Bengio (2000b) *except for the introduction of a weight-sharing scheme*: as illustrated in Fig. 20.7. The parameters of the hidden units of different groups j are shared, i.e., the weights W'_{jki} from the i-th input $\boldsymbol{x}_i$ to the k-th element of the j-th group of hidden unit h_{jk} $(j \geq i)$ are shared:

$$W'_{jki} = W_{ki}$$

with $(W_{1i}, W_{2i}, \ldots)$ denoted $\boldsymbol{W}_{:,i}$ in Fig. 20.7.

This particular sharing pattern is motivated in Larochelle and Murray (2011) by the computations performed in the mean-field inference[8] of an RBM, when only

[8] Here, unlike in Section 13.5, the inference is over some of the input variables that are missing, given the observed ones.

the first i inputs are given and one tries to infer the subsequent ones. This mean-field inference corresponds to running a recurrent network with shared weights and the first step of that inference is the same as in NADE. The only difference is that with the proposed NADE, the output weights are not forced to be simply transpose values of the input weights (they are not tied). One could imagine actually extending this procedure to not just one time step of the mean-field recurrent inference but to k steps, as in Raiko *et al.* (2014).

Although the neural auto-regressive networks and NADE were originally proposed to deal with discrete distributions, they can in principle be generalized to continuous ones by replacing the conditional discrete probability distributions (for $P(\boldsymbol{x}_j \mid \boldsymbol{x}_{j-1}, \ldots, \boldsymbol{x}_1)$) by continuous ones and following general practice to predict continuous random variables with neural networks (see Section 6.3.2) using the log-likelihood framework. A fairly generic way of parametrizing a continuous density is as a Gaussian mixture. RNADE uses this parameterization to extend NADE to real values. Stochastic gradient descent can be numerically ill-behaved due to the interactions between the conditional means and the conditional variances. The gradient on the mean is divided by the conditional variance, so the gradient can become large when the variances become small. Uria *et al.* (2013) have used a heuristic to rescale the gradient on the component means by the associated standard deviation which seems to have helped optimizing RNADE.

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

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

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

In the same paper, the authors propose to consider deep versions of the architecture, but unfortunately that immediately makes computation as expensive as in the original neural auto-regressive neural network (Bengio and Bengio, 2000b).

The first layer and the output layer can still be computed in $O(nh)$ multiply-add operations, as in the regular NADE, where h is the number of hidden units (the size of the groups h_i, in Figures 20.7 and 20.6), whereas it is $O(n^2h)$ in Bengio and Bengio (2000b). However, for the other hidden layers, the computation is $O(n^2h^2)$ if every "previous" group at layer l participates in predicting the "next" group at layer $l+1$, assuming n groups of h hidden units at each layer. Making the i-th group at layer $l+1$ only depend on the i-th group, as in Murray and Larochelle (2014) at layer l reduces it to $O(nh^2)$, which is still h times worse than the regular NADE.

20.11 A Generative View of Autoencoders

Many kinds of autoencoders can be viewed as probabilistic models. Different autoencoders can be interpreted as probabilistic models in different ways.

One of the first probabilistic interpretations of autoencoders was the view denoising autoencoders as energy-based models trained using regularized score matching. See Sections 15.9.1 and 18.5 for details. Since the early work (Vincent, 2011a) made the connection with Gaussian RBMs, this gave denoising auto-encoders with a particular parametrization a generative interpretation (they could be sampled from using the MCMC sampling techniques for Gaussian RBMs) .

The next milestone in connecting auto-encoders with a generative interpretation came with the work of Rifai *et al.* (2012). It relied on the view of contractive auto-encoders as estimators of the *tangent of the manifold* near which probability concentrates, discussed in Section 15.10 (see also Figures 15.9, 17.3). In this context, Rifai *et al.* (2012) demonstrated experimentally that good samples could be obtained from a trained contractive auto-encoder by alternating encoding, decoding, and adding noise in a particular way.

As discussed in Section 15.9.1, the application of the encoder/decoder pair moves the input configuration towards a more probable one. This can be exploited to actually sample from the estimated distribution. If you consider most Monte-Carlo Markov Chain (MCMC) algorithms, they have two elements:

1. move from lower probability configurations towards higher probability configurations, and

2. inject randomness so that the chain moves around (and does not stay stuck at some peak of probability, or mode) and has a chance to visit every configuration in the whole space, with a relative frequency equal to its probability under the underlying model.

So conceptually all one needs to do is to perform encode-decode operations (go towards more probable configurations) as well as inject noise (to move around the

probable configurations), as hinted at in (Mesnil *et al.*, 2012; Rifai *et al.*, 2012).

20.11.1 Markov Chain Associated with any Denoising Auto-Encoder

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

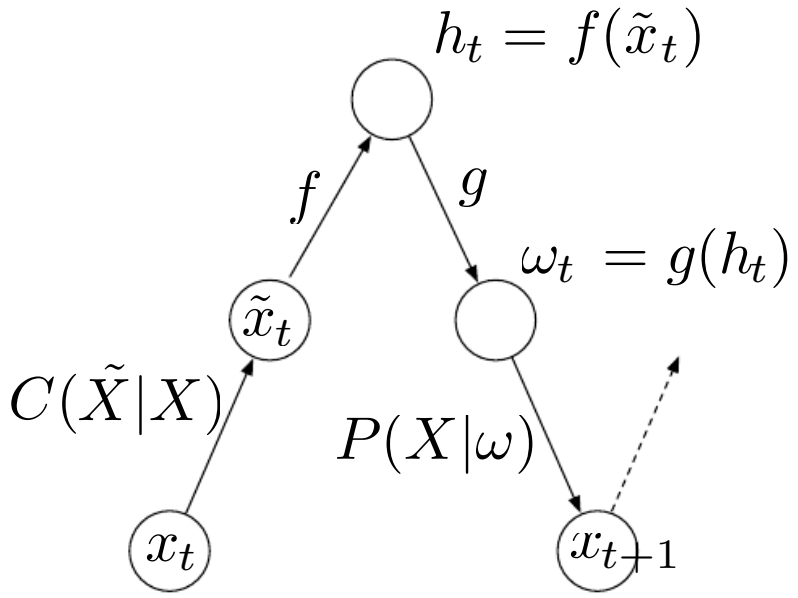

Figure 20.8: Each step of the Markov chain associated with a trained denoising auto-encoder, that generates the samples from the probabilistic model implicitly trained by the denoising reconstruction criterion. Each step consists in (a) injecting corruption C in state $\boldsymbol{x}$, yielding $\tilde{\boldsymbol{x}}$, (b) encoding it with f, yielding $\boldsymbol{h} = f(\tilde{\boldsymbol{x}})$, (c) decoding the result with g, yielding parameters ω for the reconstruction distribution, and (d) given ω, sampling a new state from the reconstruction distribution $P(\mathbf{x} \mid \omega = g(f(\tilde{\boldsymbol{x}})))$. In the typical squared reconstruction error case, $g(\boldsymbol{h}) = \hat{\boldsymbol{x}}$, which estimates $E[\boldsymbol{x} \mid \tilde{\boldsymbol{x}}]$, corruption consists in adding Gaussian noise and sampling from $P(\mathbf{x} \mid \omega)$ consists in adding another Gaussian noise to the reconstruction $\hat{\boldsymbol{x}}$. The latter noise level should correspond to the mean squared error of reconstructions, whereas the injected noise is a hyperparameter that controls the mixing speed as well as the extent to which the estimator *smoothes* the empirical distribution (Vincent, 2011b). In the figure, only the C and P conditionals are stochastic steps (f and g are deterministic computations), although noise can also be injected inside the auto-encoder, as in generative stochastic networks (Bengio *et al.*, 2014b)

Each step of the Markov chain that generates from the estimated distribution

consists of the following sub-steps, illustrated in Figure 20.8:

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

The theorem states that if the auto-encoder $P(\mathbf{x} \mid \tilde{\mathbf{x}})$ forms a consistent estimator of corresponding true conditional distribution, then the stationary distribution of the above Markov chain forms a consistent estimator (albeit an implicit one) of the data generating distribution of $\mathbf{x}$.

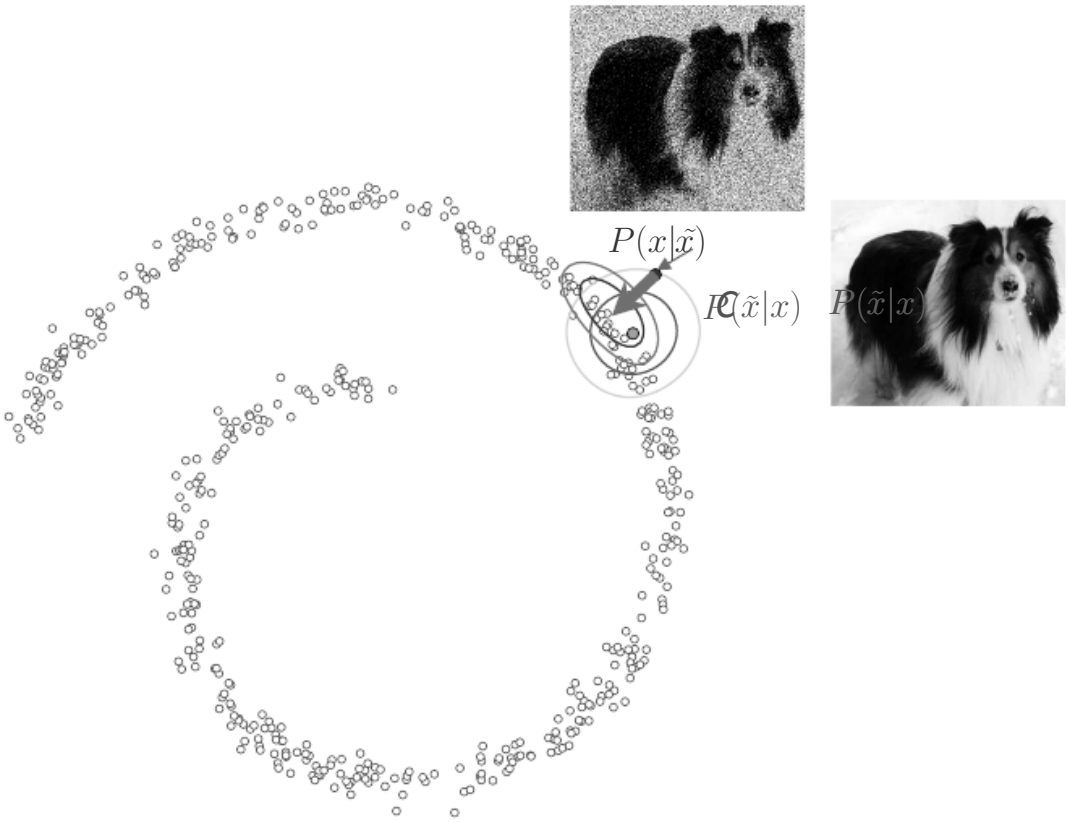

Figure 20.9: Illustration of one step of the sampling Markov chain associated with a denoising auto-encoder (see also Figure 20.8). In the figure, the data (black circles) are sitting near a low-dimensional manifold (a spiral, here), and the two stochastic steps of the Markov chain are first to corrupt $\boldsymbol{x}$ (clean image of dog, green circle) into $\tilde{\boldsymbol{x}}$ (noisy image of dog, blue circle) via $C(\tilde{\mathbf{x}} \mid \boldsymbol{x})$ (here an isotropic Gaussian noise in green), and then to sample a new $\boldsymbol{x}$ via the estimated denoising $P(\boldsymbol{x} \mid \tilde{\boldsymbol{x}})$. Note how there are many possible $\boldsymbol{x}$ which could have given rise to $\tilde{\boldsymbol{x}}$, and these all lie on the manifold in the neighborhood of $\tilde{\boldsymbol{x}}$, hence the flattened shape of $P(\mathbf{x} \mid \tilde{\boldsymbol{x}})$ (in blue). *Modified from a figure first created and graciously authorized by Jason Yosinski.*

Figure 20.9 illustrates the sampling process of the DAE in a way that complements Figure 20.8, with a specific imagined example. For a more elaborate discussion of the probabilistic nature of denoising auto-encoders, and their generalization (Bengio *et al.*, 2014b), *Generative Stochastic Networks* (GSNs), see Section 20.12 below. In particular, the noise does not have to be injected only in the input, and it could be injected anywhere along the chain. GSNs also generalize DAEs by allowing the state of the Markov chain to be extended beyond the visible variable $\boldsymbol{x}$, to include also some latent variable $\boldsymbol{h}$. Finally, Section 20.12 discusses training strategies for DAEs that are aimed at making it a better generative model and not just a good feature learner.

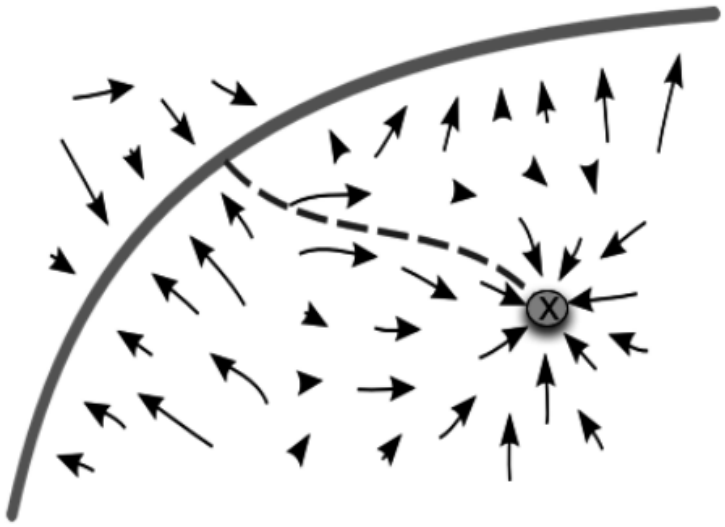

Figure 20.10: Illustration of the effect of the walk-back training procedure, used for denoising auto-encoders or GSNs in general. The objective is to remove spurious modes faster by letting the Markov chain go towards them (along the red path, starting on the purple data manifold and following the arrows plus noise), and then punishing the Markov chain for this behavior (i.e., walking back to the right place) by telling the chain to return towards the data manifold (reconstruct the original data).

20.11.2 Clamping and Conditional Sampling

Similarly to Boltzmann machines, denoising auto-encoders and GSNs can be used to sample from a conditional distribution $P(\mathbf{x}_f \mid \mathbf{x}_o)$, simply by clamping the *observed* units $\mathbf{x}_f$ and only resampling the *free* units $\mathbf{x}_o$ given $\mathbf{x}_f$ and the sampled latent variables (if any). This has been introduced by Bengio *et al.* (2014b).

However, note that Proposition 1 of that paper is missing a condition: the transition operator (defined by the stochastic mapping going from one state of the chain to the next) should satisfy *detailed balance*, described in Section 14.1.1.

An experiment in clamping half of the pixels (the right part of the image) and running the Markov chain on the other half is shown in Figure 20.11.

Figure 20.11: Illustration of clamping the right half of the image and running the Markov by resampling only the left half at each step. These samples come from a GSN trained to reconstruct MNIST digits at each time step, i.e., using the walkback procedure.

20.11.3 Walk-Back Training Procedure

The walk-back training procedure was proposed by Bengio *et al.* (2013b) as a way to speed-up the convergence of generative training of denoising auto-encoders. Instead of performing a one-step encode-decode reconstruction, this procedure consists in alternative multiple stochastic encode-decode steps (as in the generative Markov chain) initialized at a training example (just like with the contrastive divergence algorithm, described in Sections 18.2) and 20.3.1) and penalizing the last probabilistic reconstructions (or all of the reconstructions along the way).

It was shown in that paper that training with k steps is equivalent (in the sense of achieving the same stationary distribution) as training with one step, but practically has the advantage that spurious modes farther from the data can be removed more efficiently, as illustrated in Figure 20.10.

Figure 20.12 illustrates the application of the walk-back procedure in a generative stochastic network, which is described in the next section.

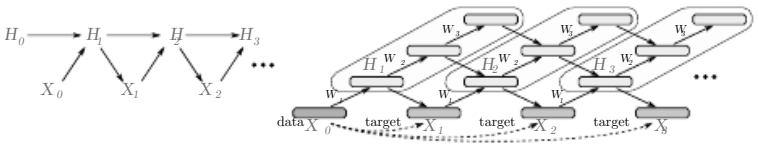

Figure 20.12: Left: graphical model of the generative Markov chain associated with a generative stochastic network (GSN). Right specific case where the latent variable is formed of several layers, each connected to the one above and the one below, making the generative process very similar to Gibbs sampling in a deep Boltzmann machine (Salakhutdinov and Hinton, 2009b). The walk-back training procedure is used, i.e., at every step the reconstruction probability distribution is pushed towards generating the training example (which also initializes the chain).

20.12 Generative Stochastic Networks

Generative stochastic networks (Bengio *et al.*, 2014b) or GSNs are generalizations of denoising auto-encoders that include latent variables in the generative Markov chain, in addition to the visible variables (usually denoted $\boldsymbol{x}$). The generative Markov chain looks like the one in Figure 20.13. An example of a GSN structured like a deep Boltzmann machine and trained by the walk-back procedure is shown in Figure 20.12.

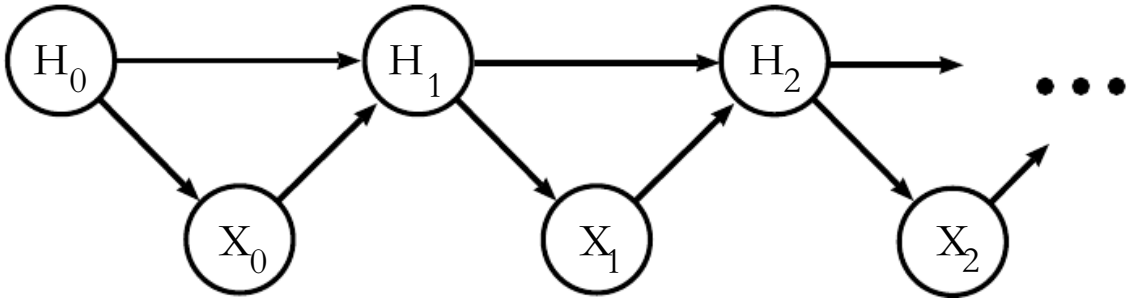

Figure 20.13: Markov chain of a GSN (Generative Stochastic Network) with latent variables with H and visible variable X, i.e., an unfolding of the generative process with X_k and H_k at step k of the chain. TODO: please use h and x etc. throughout the GSN section

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

1. $P(X_k \mid H_k)$ tells how to generate the next visible variable given the current latent state. Such a "reconstruction distribution" is also found in denoising auto-encoders, RBMs, DBNs and DBMs.

2. $P(H_k \mid H_{k-1}, X_{k-1})$ tells how to update the latent state variable, given the previous latent state and visible variable.

Denoising auto-encoders and GSNs differ from classical probabilistic models (directed or undirected) in that it parametrizes the generative process itself rather than the mathematical specification of the joint distribution of visible and latent variables. Instead, the latter is defined *implicitly*, *if it exists*, as the stationary distribution of the generative Markov chain. The conditions for existence of the stationary distribution are mild (basically, the chain mixes) but can be violated by some choices of the transition distributions (for example, if they were deterministic).

One could imagine different training criteria for GSNs. The one proposed and evaluated by Bengio *et al.* (2014b) is simply reconstruction log-probability on the visible units, just like for denoising auto-encoders. This is achieved by clamping $X_0 = x$ to the observed example and maximizing the probability of generating x at some subsequent time steps, i.e., maximizing $\log P(X_k = x \mid H_k)$, where H_k is sampled from the chain, given $X_0 = x$. In order to estimate the gradient of $\log P(X_k = x \mid H_k)$ with respect to the other pieces of the model, Bengio *et al.* (2014b) use the reparametrization trick, introduced in Section 13.5.1.

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

20.12.1 Discriminant GSNs

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

For example, Zhou and Troyanskaya (2014) generalize GSNs in this way, by only back-propagating the reconstruction log-probability over the output variables, keeping the input variables fixed. They applied this successfully to model *sequences* (protein secondary structure) and introduced a (one-dimensional) *convolutional* structure in the transition operator of the Markov chain. Keep in mind that, for each step of the Markov chain, one generates a new sequence for each layer, and that sequence is the input for computing other layer values (say the one below and the one above) at the next time step, as illustrated in Figure 20.14.

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

Zöhrer and Pernkopf (2014) considered a hybrid model that combines a supervised objective (as in the above work) and an unsupervised objective (as in

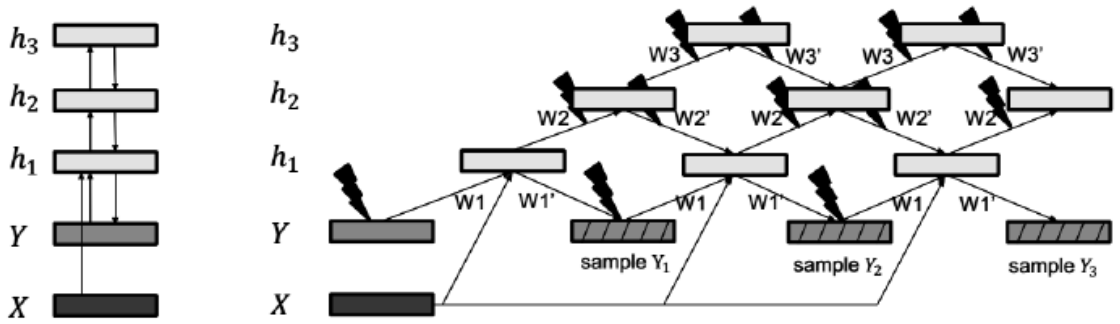

Figure 20.14: Markov chain arising out of a discriminant GSN, i.e., where a GSN is used as a structured output model over a variable y, conditioned on an input X. Reproduced with permission from Zhou and Troyanskaya (2014). The structure is as in a GSN (over the output) but with computations being conditioned on the input X at each step.

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

20.13 Methodological Notes

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

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

Another subtletly of evaluating generative models is that the evaluation metrics are often hard research problems in and of themselves. It can be very difficult to establish that models are being compared fairly. For example, suppose we use

AIS to estimate $\log Z$ in order to compute $\log \tilde{p}(\boldsymbol{x}) - \log Z$ for a new model we have just invented. A computationally economical implementation of AIS may fail to find several modes of the model distribution and underestimate Z, which will result in us overestimating $\log p(\boldsymbol{x})$. It can thus be difficult to tell whether a good likelihood estimate is due to a good model or a bad AIS implementation.

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

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

Finally, in some cases the likelihood seems not to measure any attribute of the model that we really care about. For example, real-valued models of MNIST can obtain arbitrarily high likelihood by assigning arbitrarily low variance to background pixels that never change. Models and algorithms that detect these constant features can reap unlimited rewards, even though this is not a very useful

thing to do. This strongly suggests a need for developing other ways of evaluating generative models.

Although this is still an open question, this might be achieved by converting the problem into a classification task. For example, we have seen that the NCE method (Noise Contrastive Estimation, Section 18.6) compares the density of the training data according to a learned unnormalized model with its density under a background model. However, generative models do not always provide us with an energy function (equivalently, an unnormalized density), e.g., deep Boltzmann machines, generative stochastic networks, most denoising auto-encoders (that are not guaranteed to correspond to an energy function), deep Belief networks, etc. Therefore, it would be interesting to consider a classification task in which one tries to distinguish the training examples from the generated examples. This is precisely what is achieved by the discriminator network of generative adversarial networks (Section 20.9.5). However, it would require an expensive operation (training a discriminator) each time one would have to evaluate performance

Algorithm 20.3 The variational stochastic maximum likelihood algorithm for training a 2 hidden-layer DBM.

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 $p(\boldsymbol{v}, \boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)}; \boldsymbol{\theta} + \epsilon \Delta_{\boldsymbol{\theta}})$ toburn in, starting from samples from $p(\boldsymbol{v}, \boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)}; \boldsymbol{\theta})$.
Initialize three random matrices, $\tilde{\boldsymbol{V}}$, $\tilde{\boldsymbol{H}}^{(1)}$ and $\tilde{\boldsymbol{H}}^{(2)}$ each with m columns 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 columns of a matrix $\boldsymbol{V} = \left[\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(m)}\right]$ from the training set.
  while Not converged (Mean-field inference loop) **do**
    $\tilde{\boldsymbol{H}}^{(1)} \leftarrow \text{sigmoid}\left(\boldsymbol{V}^{\top}\boldsymbol{W}^{(1)} + \tilde{\boldsymbol{H}}^{(2)\top}\boldsymbol{W}^{(2)\top}\right)$.
    $\tilde{\boldsymbol{H}}^{(2)} \leftarrow \text{sigmoid}\left(\tilde{\boldsymbol{H}}^{(1)}\boldsymbol{W}^{(2)}\right)$.
  end while
  $\Delta_{\boldsymbol{W}^{(1)}} \leftarrow \frac{1}{m}\boldsymbol{V}\hat{\boldsymbol{H}}^{(1)\top}$
  $\Delta_{\boldsymbol{W}^{(2)}} \leftarrow \frac{1}{m}\hat{\boldsymbol{H}}^{(1)}\hat{\boldsymbol{H}}^{(2)\top}$
  for $l = 1$ to k (Gibbs sampling) **do**
    Gibbs block 1:
    $\tilde{\boldsymbol{V}}$sampled from $\prod_{i=1}^{m}\prod_{a=1}^{d}$sigmoid $\left(\boldsymbol{W}^{(1)}_{a,:}\tilde{\boldsymbol{H}}^{(2)}_{:,i}\right)$.
    $\tilde{\boldsymbol{H}}^{(2)}$sampled from $\prod_{i=1}^{m}\prod_{b=1}^{m}$sigmoid $\left(\tilde{\boldsymbol{H}}^{(1)\top}_{:,i}\boldsymbol{W}^{(2)}_{:,b}\right)$.
    Gibbs block 2:
    $\tilde{\boldsymbol{H}}^{(1)}$sampled from $\prod_{i=1}^{m}\prod_{j=1}^{n}$sigmoid $\left(\tilde{\boldsymbol{V}}^{\top}_{:,i}\boldsymbol{W}^{(1)}_{:,j} + \boldsymbol{W}^{(2)}_{j,:}\tilde{\boldsymbol{H}}^{(2)}_{:,i}\right)$.
  end for
  $\Delta_{\boldsymbol{W}^{(1)}} \leftarrow \Delta_{\boldsymbol{W}^{(1)}} - \frac{1}{m}\boldsymbol{V}\hat{\boldsymbol{H}}^{(1)\top}$
  $\Delta_{\boldsymbol{W}^{(2)}} \leftarrow \Delta_{\boldsymbol{W}^{(2)}} - \frac{1}{m}\hat{\boldsymbol{H}}^{(1)}\hat{\boldsymbol{H}}^{(2)\top}$
  $\boldsymbol{W}^{(1)} \leftarrow \boldsymbol{W}^{(1)} + \epsilon\Delta_{\boldsymbol{W}^{(1)}}$
  $\boldsymbol{W}^{(2)} \leftarrow \boldsymbol{W}^{(2)} + \epsilon\Delta_{\boldsymbol{W}^{(2)}}$
end while