Part III

Deep Learning Research

This part of the book describes the more ambitious and advanced approaches to deep learning, currently pursued by the research community.

In the previous parts of the book, we have shown how to solve supervised learning problems—how to learn to map one vector to another, given enough examples of the mapping.

Not all problems we might want to solve fall into this category. We may wish to generate new examples, or determine how likely some point is, or handle missing values and take advantage of a large set of unlabeled examples or examples from related tasks. A shortcoming of the current state of the art for industrial applications is that our learning algorithms require large amounts of supervised data to achieve good accuracy. In this part of the book, we discuss some of the speculative approaches to reducing the amount of labeled data necessary for existing models to work well and be applicable across a broader range of tasks. Accomplishing these goals usually requires some form of unsupervised or semi-supervised learning.

Many deep learning algorithms have been designed to tackle unsupervised learning problems, but none have truly solved the problem in the same way that deep learning has largely solved the supervised learning problem for a wide variety of tasks. In this part of the book, we describe the existing approaches to unsupervised learning and some of the popular thought about how we can make progress in this field.

A central cause of the difficulties with unsupervised learning is the high dimensionality of the random variables being modeled. This brings two distinct challenges: a statistical challenge and a computational challenge. The *statistical challenge* regards generalization: the number of configurations we may want to distinguish can grow exponentially with the number of dimensions of interest, and this quickly becomes much larger than the number of examples one can possibly have (or use with bounded computational resources). The *computational challenge* associated with high-dimensional distributions arises because many algorithms for learning or using a trained model (especially those based on estimating an explicit probability function) involve intractable computations that grow exponentially with the number of dimensions.

With probabilistic models, this computational challenge arises from the need to perform intractable inference or simply from the need to normalize the distribution.

- *Intractable inference*: inference is discussed mostly in chapter 19. It regards the question of guessing the probable values of some variables a, given other variables b, with respect to a model that captures the joint distribution over

a, b and c. In order to even compute such conditional probabilities one needs to sum over the values of the variables c, as well as compute a normalization constant which sums over the values of a and c.

- *Intractable normalization constants (the partition function)*: the partition function is discussed mostly in chapter 18. Normalizing constants of probability functions come up in inference (above) as well as in learning. Many probabilistic models involve such a normalizing constant. Unfortunately, learning such a model often requires computing the gradient of the logarithm of the partition function with respect to the model parameters. That computation is generally as intractable as computing the partition function itself. Monte Carlo Markov chain (MCMC) methods (chapter 17) are often used to deal with the partition function (computing it or its gradient). Unfortunately, MCMC methods suffer when the modes of the model distribution are numerous and well-separated, especially in high-dimensional spaces (section 17.5).

One way to confront these intractable computations is to approximate them, and many approaches have been proposed as discussed in this third part of the book. Another interesting way, also discussed here, would be to avoid these intractable computations altogether by design, and methods that do not require such computations are thus very appealing. Several generative models have been proposed in recent years, with that motivation. A wide variety of contemporary approaches to generative modeling are discussed in chapter 20.

Part III is the most important for a researcher—someone who wants to understand the breadth of perspectives that have been brought to the field of deep learning, and push the field forward towards true artificial intelligence.

Chapter 13

Linear Factor Models

Many of the research frontiers in deep learning involve building a probabilistic model of the input, $p_{\text{model}}(\boldsymbol{x})$. Such a model can, in principle, use probabilistic inference to predict any of the variables in its environment given any of the other variables. Many of these models also have latent variables $\boldsymbol{h}$, with $p_{\text{model}}(\boldsymbol{x}) = \mathbb{E}_{\boldsymbol{h}} p_{\text{model}}(\boldsymbol{x} \mid \boldsymbol{h})$. These latent variables provide another means of representing the data. Distributed representations based on latent variables can obtain all of the advantages of representation learning that we have seen with deep feedforward and recurrent networks.

In this chapter, we describe some of the simplest probabilistic models with latent variables: linear factor models. These models are sometimes used as building blocks of mixture models (Hinton *et al.*, 1995a; Ghahramani and Hinton, 1996; Roweis *et al.*, 2002) or larger, deep probabilistic models (Tang *et al.*, 2012). They also show many of the basic approaches necessary to build generative models that the more advanced deep models will extend further.

A linear factor model is defined by the use of a stochastic, linear decoder function that generates $\boldsymbol{x}$ by adding noise to a linear transformation of $\boldsymbol{h}$.

These models are interesting because they allow us to discover explanatory factors that have a simple joint distribution. The simplicity of using a linear decoder made these models some of the first latent variable models to be extensively studied.

A linear factor model describes the data generation process as follows. First, we sample the explanatory factors $\boldsymbol{h}$ from a distribution

$$\mathbf{h} \sim p(\boldsymbol{h}), \tag{13.1}$$

where $p(\boldsymbol{h})$ is a factorial distribution, with $p(\boldsymbol{h}) = \prod_i p(h_i)$, so that it is easy to

sample from. Next we sample the real-valued observable variables given the factors:

$$\boldsymbol{x} = \boldsymbol{W}\boldsymbol{h} + \boldsymbol{b} + \text{noise} \tag{13.2}$$

where the noise is typically Gaussian and diagonal (independent across dimensions). This is illustrated in figure 13.1.

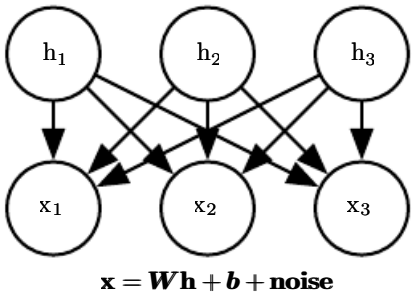


Figure 13.1: The directed graphical model describing the linear factor model family, in which we assume that an observed data vector $\boldsymbol{x}$ is obtained by a linear combination of independent latent factors $\boldsymbol{h}$, plus some noise. Different models, such as probabilistic PCA, factor analysis or ICA, make different choices about the form of the noise and of the prior $p(\boldsymbol{h})$.

13.1 Probabilistic PCA and Factor Analysis

Probabilistic PCA (principal components analysis), factor analysis and other linear factor models are special cases of the above equations (13.1 and 13.2) and only differ in the choices made for the noise distribution and the model's prior over latent variables $\boldsymbol{h}$ before observing $\boldsymbol{x}$.

In **factor analysis** (Bartholomew, 1987; Basilevsky, 1994), the latent variable prior is just the unit variance Gaussian

$$\mathbf{h} \sim \mathcal{N}(\boldsymbol{h}; \mathbf{0}, \boldsymbol{I}) \tag{13.3}$$

while the observed variables x_i are assumed to be **conditionally independent**, given $\boldsymbol{h}$. Specifically, the noise is assumed to be drawn from a diagonal covariance Gaussian distribution, with covariance matrix $\boldsymbol{\psi} = \text{diag}(\boldsymbol{\sigma}^2)$, with $\boldsymbol{\sigma}^2 = [\sigma_1^2, \sigma_2^2, \ldots, \sigma_n^2]^\top$ a vector of per-variable variances.

The role of the latent variables is thus to *capture the dependencies* between the different observed variables x_i. Indeed, it can easily be shown that $\boldsymbol{x}$ is just a multivariate normal random variable, with

$$\mathbf{x} \sim \mathcal{N}(\boldsymbol{x}; \boldsymbol{b}, \boldsymbol{W}\boldsymbol{W}^\top + \boldsymbol{\psi}). \tag{13.4}$$

In order to cast PCA in a probabilistic framework, we can make a slight modification to the factor analysis model, making the conditional variances σ_i^2 equal to each other. In that case the covariance of $\boldsymbol{x}$ is just $\boldsymbol{W}\boldsymbol{W}^\top + \sigma^2\boldsymbol{I}$, where σ^2 is now a scalar. This yields the conditional distribution

$$\mathbf{x} \sim \mathcal{N}(\boldsymbol{x}; \boldsymbol{b}, \boldsymbol{W}\boldsymbol{W}^\top + \sigma^2\boldsymbol{I}) \tag{13.5}$$

or equivalently

$$\mathbf{x} = \boldsymbol{W}\mathbf{h} + \boldsymbol{b} + \sigma\mathbf{z} \tag{13.6}$$

where $\mathbf{z} \sim \mathcal{N}(\boldsymbol{z}; \mathbf{0}, \boldsymbol{I})$ is Gaussian noise. Tipping and Bishop (1999) then show an iterative EM algorithm for estimating the parameters $\boldsymbol{W}$ and σ^2.

This **probabilistic PCA** model takes advantage of the observation that most variations in the data can be captured by the latent variables $\boldsymbol{h}$, up to some small residual **reconstruction error** σ^2. As shown by Tipping and Bishop (1999), probabilistic PCA becomes PCA as $\sigma \to 0$. In that case, the conditional expected value of $\boldsymbol{h}$ given $\boldsymbol{x}$ becomes an orthogonal projection of $\boldsymbol{x} - \boldsymbol{b}$ onto the space spanned by the d columns of $\boldsymbol{W}$, like in PCA.

As $\sigma \to 0$, the density model defined by probabilistic PCA becomes very sharp around these d dimensions spanned by the columns of $\boldsymbol{W}$. This can make the model assign very low likelihood to the data if the data does not actually cluster near a hyperplane.

13.2 Independent Component Analysis (ICA)

Independent component analysis (ICA) is among the oldest representation learning algorithms (Herault and Ans, 1984; Jutten and Herault, 1991; Comon, 1994; Hyvärinen, 1999; Hyvärinen *et al.*, 2001a; Hinton *et al.*, 2001; Teh *et al.*, 2003). It is an approach to modeling linear factors that seeks to separate an observed signal into many underlying signals that are scaled and added together to form the observed data. These signals are intended to be fully independent, rather than merely decorrelated from each other.[1]

Many different specific methodologies are referred to as ICA. The variant that is most similar to the other generative models we have described here is a variant (Pham *et al.*, 1992) that trains a fully parametric generative model. The prior distribution over the underlying factors, $p(\boldsymbol{h})$, must be fixed ahead of time by the user. The model then deterministically generates $\boldsymbol{x} = \boldsymbol{W}\boldsymbol{h}$. We can perform a

[1] See section 3.8 for a discussion of the difference between uncorrelated variables and independent variables.

nonlinear change of variables (using equation 3.47) to determine $p(\boldsymbol{x})$. Learning the model then proceeds as usual, using maximum likelihood.

The motivation for this approach is that by choosing $p(\boldsymbol{h})$ to be independent, we can recover underlying factors that are as close as possible to independent. This is commonly used, not to capture high-level abstract causal factors, but to recover low-level signals that have been mixed together. In this setting, each training example is one moment in time, each x_i is one sensor's observation of the mixed signals, and each h_i is one estimate of one of the original signals. For example, we might have n people speaking simultaneously. If we have n different microphones placed in different locations, ICA can detect the changes in the volume between each speaker as heard by each microphone, and separate the signals so that each h_i contains only one person speaking clearly. This is commonly used in neuroscience for electroencephalography, a technology for recording electrical signals originating in the brain. Many electrode sensors placed on the subject's head are used to measure many electrical signals coming from the body. The experimenter is typically only interested in signals from the brain, but signals from the subject's heart and eyes are strong enough to confound measurements taken at the subject's scalp. The signals arrive at the electrodes mixed together, so ICA is necessary to separate the electrical signature of the heart from the signals originating in the brain, and to separate signals in different brain regions from each other.

As mentioned before, many variants of ICA are possible. Some add some noise in the generation of $\boldsymbol{x}$ rather than using a deterministic decoder. Most do not use the maximum likelihood criterion, but instead aim to make the elements of $\boldsymbol{h} = \boldsymbol{W}^{-1}\boldsymbol{x}$ independent from each other. Many criteria that accomplish this goal are possible. Equation 3.47 requires taking the determinant of $\boldsymbol{W}$, which can be an expensive and numerically unstable operation. Some variants of ICA avoid this problematic operation by constraining $\boldsymbol{W}$ to be orthogonal.

All variants of ICA require that $p(\boldsymbol{h})$ be non-Gaussian. This is because if $p(\boldsymbol{h})$ is an independent prior with Gaussian components, then $\boldsymbol{W}$ is not identifiable. We can obtain the same distribution over $p(\boldsymbol{x})$ for many values of $\boldsymbol{W}$. This is very different from other linear factor models like probabilistic PCA and factor analysis, that often require $p(\boldsymbol{h})$ to be Gaussian in order to make many operations on the model have closed form solutions. In the maximum likelihood approach where the user explicitly specifies the distribution, a typical choice is to use $p(h_i) = \frac{d}{dh_i}\sigma(h_i)$. Typical choices of these non-Gaussian distributions have larger peaks near 0 than does the Gaussian distribution, so we can also see most implementations of ICA as learning sparse features.

Many variants of ICA are not generative models in the sense that we use the phrase. In this book, a generative model either represents $p(\boldsymbol{x})$ or can draw samples from it. Many variants of ICA only know how to transform between $\boldsymbol{x}$ and $\boldsymbol{h}$, but do not have any way of representing $p(\boldsymbol{h})$, and thus do not impose a distribution over $p(\boldsymbol{x})$. For example, many ICA variants aim to increase the sample kurtosis of $\boldsymbol{h} = \boldsymbol{W}^{-1}\boldsymbol{x}$, because high kurtosis indicates that $p(\boldsymbol{h})$ is non-Gaussian, but this is accomplished without explicitly representing $p(\boldsymbol{h})$. This is because ICA is more often used as an analysis tool for separating signals, rather than for generating data or estimating its density.

Just as PCA can be generalized to the nonlinear autoencoders described in chapter 14, ICA can be generalized to a nonlinear generative model, in which we use a nonlinear function f to generate the observed data. See Hyvärinen and Pajunen (1999) for the initial work on nonlinear ICA and its successful use with ensemble learning by Roberts and Everson (2001) and Lappalainen *et al.* (2000). Another nonlinear extension of ICA is the approach of **nonlinear independent components estimation**, or NICE (Dinh *et al.*, 2014), which stacks a series of invertible transformations (encoder stages) that have the property that the determinant of the Jacobian of each transformation can be computed efficiently. This makes it possible to compute the likelihood exactly and, like ICA, attempts to transform the data into a space where it has a factorized marginal distribution, but is more likely to succeed thanks to the nonlinear encoder. Because the encoder is associated with a decoder that is its perfect inverse, it is straightforward to generate samples from the model (by first sampling from $p(\boldsymbol{h})$ and then applying the decoder).

Another generalization of ICA is to learn groups of features, with statistical dependence allowed within a group but discouraged between groups (Hyvärinen and Hoyer, 1999; Hyvärinen *et al.*, 2001b). When the groups of related units are chosen to be non-overlapping, this is called **independent subspace analysis**. It is also possible to assign spatial coordinates to each hidden unit and form overlapping groups of spatially neighboring units. This encourages nearby units to learn similar features. When applied to natural images, this **topographic ICA** approach learns Gabor filters, such that neighboring features have similar orientation, location or frequency. Many different phase offsets of similar Gabor functions occur within each region, so that pooling over small regions yields translation invariance.

13.3 Slow Feature Analysis

Slow feature analysis (SFA) is a linear factor model that uses information from

time signals to learn invariant features (Wiskott and Sejnowski, 2002).

Slow feature analysis is motivated by a general principle called the slowness principle. The idea is that the important characteristics of scenes change very slowly compared to the individual measurements that make up a description of a scene. For example, in computer vision, individual pixel values can change very rapidly. If a zebra moves from left to right across the image, an individual pixel will rapidly change from black to white and back again as the zebra's stripes pass over the pixel. By comparison, the feature indicating whether a zebra is in the image will not change at all, and the feature describing the zebra's position will change slowly. We therefore may wish to regularize our model to learn features that change slowly over time.

The slowness principle predates slow feature analysis and has been applied to a wide variety of models (Hinton, 1989; Földiák, 1989; Mobahi *et al.*, 2009; Bergstra and Bengio, 2009). In general, we can apply the slowness principle to any differentiable model trained with gradient descent. The slowness principle may be introduced by adding a term to the cost function of the form

$$\lambda \sum_t L(f(\boldsymbol{x}^{(t+1)}), f(\boldsymbol{x}^{(t)})) \tag{13.7}$$

where λ is a hyperparameter determining the strength of the slowness regularization term, t is the index into a time sequence of examples, f is the feature extractor to be regularized, and L is a loss function measuring the distance between $f(\boldsymbol{x}^{(t)})$ and $f(\boldsymbol{x}^{(t+1)})$. A common choice for L is the mean squared difference.

Slow feature analysis is a particularly efficient application of the slowness principle. It is efficient because it is applied to a linear feature extractor, and can thus be trained in closed form. Like some variants of ICA, SFA is not quite a generative model per se, in the sense that it defines a linear map between input space and feature space but does not define a prior over feature space and thus does not impose a distribution $p(\boldsymbol{x})$ on input space.

The SFA algorithm (Wiskott and Sejnowski, 2002) consists of defining $f(\boldsymbol{x}; \boldsymbol{\theta})$ to be a linear transformation, and solving the optimization problem

$$\min_{\boldsymbol{\theta}} \mathbb{E}_t (f(\boldsymbol{x}^{(t+1)})_i - f(\boldsymbol{x}^{(t)})_i)^2 \tag{13.8}$$

subject to the constraints

$$\mathbb{E}_t f(\boldsymbol{x}^{(t)})_i = 0 \tag{13.9}$$

and

$$\mathbb{E}_t [f(\boldsymbol{x}^{(t)})_i^2] = 1. \tag{13.10}$$

The constraint that the learned feature have zero mean is necessary to make the problem have a unique solution; otherwise we could add a constant to all feature values and obtain a different solution with equal value of the slowness objective. The constraint that the features have unit variance is necessary to prevent the pathological solution where all features collapse to 0. Like PCA, the SFA features are ordered, with the first feature being the slowest. To learn multiple features, we must also add the constraint

$$\forall i < j, \mathbb{E}_t[f(\boldsymbol{x}^{(t)})_i f(\boldsymbol{x}^{(t)})_j] = 0. \tag{13.11}$$

This specifies that the learned features must be linearly decorrelated from each other. Without this constraint, all of the learned features would simply capture the one slowest signal. One could imagine using other mechanisms, such as minimizing reconstruction error, to force the features to diversify, but this decorrelation mechanism admits a simple solution due to the linearity of SFA features. The SFA problem may be solved in closed form by a linear algebra package.

SFA is typically used to learn nonlinear features by applying a nonlinear basis expansion to $\boldsymbol{x}$ before running SFA. For example, it is common to replace $\boldsymbol{x}$ by the quadratic basis expansion, a vector containing elements $x_i x_j$ for all i and j. Linear SFA modules may then be composed to learn deep nonlinear slow feature extractors by repeatedly learning a linear SFA feature extractor, applying a nonlinear basis expansion to its output, and then learning another linear SFA feature extractor on top of that expansion.

When trained on small spatial patches of videos of natural scenes, SFA with quadratic basis expansions learns features that share many characteristics with those of complex cells in V1 cortex (Berkes and Wiskott, 2005). When trained on videos of random motion within 3-D computer rendered environments, deep SFA learns features that share many characteristics with the features represented by neurons in rat brains that are used for navigation (Franzius *et al.*, 2007). SFA thus seems to be a reasonably biologically plausible model.

A major advantage of SFA is that it is possibly to theoretically predict which features SFA will learn, even in the deep, nonlinear setting. To make such theoretical predictions, one must know about the dynamics of the environment in terms of configuration space (e.g., in the case of random motion in the 3-D rendered environment, the theoretical analysis proceeds from knowledge of the probability distribution over position and velocity of the camera). Given the knowledge of how the underlying factors actually change, it is possible to analytically solve for the optimal functions expressing these factors. In practice, experiments with deep SFA applied to simulated data seem to recover the theoretically predicted functions.

This is in comparison to other learning algorithms where the cost function depends highly on specific pixel values, making it much more difficult to determine what features the model will learn.

Deep SFA has also been used to learn features for object recognition and pose estimation (Franzius *et al.*, 2008). So far, the slowness principle has not become the basis for any state of the art applications. It is unclear what factor has limited its performance. We speculate that perhaps the slowness prior is too strong, and that, rather than imposing a prior that features should be approximately constant, it would be better to impose a prior that features should be easy to predict from one time step to the next. The position of an object is a useful feature regardless of whether the object's velocity is high or low, but the slowness principle encourages the model to ignore the position of objects that have high velocity.

13.4 Sparse Coding

Sparse coding (Olshausen and Field, 1996) is a linear factor model that has been heavily studied as an unsupervised feature learning and feature extraction mechanism. Strictly speaking, the term "sparse coding" refers to the process of inferring the value of $\boldsymbol{h}$ in this model, while "sparse modeling" refers to the process of designing and learning the model, but the term "sparse coding" is often used to refer to both.

Like most other linear factor models, it uses a linear decoder plus noise to obtain reconstructions of $\boldsymbol{x}$, as specified in equation 13.2. More specifically, sparse coding models typically assume that the linear factors have Gaussian noise with isotropic precision β:

$$p(\boldsymbol{x} \mid \boldsymbol{h}) = \mathcal{N}(\boldsymbol{x}; \boldsymbol{W}\boldsymbol{h} + \boldsymbol{b}, \frac{1}{\beta}\boldsymbol{I}). \tag{13.12}$$

The distribution $p(\boldsymbol{h})$ is chosen to be one with sharp peaks near 0 (Olshausen and Field, 1996). Common choices include factorized Laplace, Cauchy or factorized Student-t distributions. For example, the Laplace prior parametrized in terms of the sparsity penalty coefficient λ is given by

$$p(h_i) = \text{Laplace}(h_i; 0, \frac{2}{\lambda}) = \frac{\lambda}{4} e^{-\frac{1}{2}\lambda|h_i|} \tag{13.13}$$

and the Student-t prior by

$$p(h_i) \propto \frac{1}{(1 + \frac{h_i^2}{\nu})^{\frac{\nu+1}{2}}}. \tag{13.14}$$

Training sparse coding with maximum likelihood is intractable. Instead, the training alternates between encoding the data and training the decoder to better reconstruct the data given the encoding. This approach will be justified further as a principled approximation to maximum likelihood later, in section 19.3.

For models such as PCA, we have seen the use of a parametric encoder function that predicts $\boldsymbol{h}$ and consists only of multiplication by a weight matrix. The encoder that we use with sparse coding is not a parametric encoder. Instead, the encoder is an optimization algorithm, that solves an optimization problem in which we seek the single most likely code value:

$$\boldsymbol{h}^* = f(\boldsymbol{x}) = \arg\max_{\boldsymbol{h}} p(\boldsymbol{h} \mid \boldsymbol{x}). \tag{13.15}$$

When combined with equation 13.13 and equation 13.12, this yields the following optimization problem:

$$\arg\max_{\boldsymbol{h}} p(\boldsymbol{h} \mid \boldsymbol{x}) \tag{13.16}$$

$$= \arg\max_{\boldsymbol{h}} \log p(\boldsymbol{h} \mid \boldsymbol{x}) \tag{13.17}$$

$$= \arg\min_{\boldsymbol{h}} \lambda ||\boldsymbol{h}||_1 + \beta ||\boldsymbol{x} - \boldsymbol{W}\boldsymbol{h}||_2^2, \tag{13.18}$$

where we have dropped terms not depending on $\boldsymbol{h}$ and divided by positive scaling factors to simplify the equation.

Due to the imposition of an L^1 norm on $\boldsymbol{h}$, this procedure will yield a sparse $\boldsymbol{h}^*$ (See section 7.1.2).

To train the model rather than just perform inference, we alternate between minimization with respect to $\boldsymbol{h}$ and minimization with respect to $\boldsymbol{W}$. In this presentation, we treat β as a hyperparameter. Typically it is set to 1 because its role in this optimization problem is shared with λ and there is no need for both hyperparameters. In principle, we could also treat β as a parameter of the model and learn it. Our presentation here has discarded some terms that do not depend on $\boldsymbol{h}$ but do depend on β. To learn β, these terms must be included, or β will collapse to 0.

Not all approaches to sparse coding explicitly build a $p(\boldsymbol{h})$ and a $p(\boldsymbol{x} \mid \boldsymbol{h})$. Often we are just interested in learning a dictionary of features with activation values that will often be zero when extracted using this inference procedure.

If we sample $\boldsymbol{h}$ from a Laplace prior, it is in fact a zero probability event for an element of $\boldsymbol{h}$ to actually be zero. The generative model itself is not especially sparse, only the feature extractor is. Goodfellow *et al.* (2013d) describe approximate

inference in a different model family, the spike and slab sparse coding model, for which samples from the prior usually contain true zeros.

The sparse coding approach combined with the use of the non-parametric encoder can in principle minimize the combination of reconstruction error and log-prior better than any specific parametric encoder. Another advantage is that there is no generalization error to the encoder. A parametric encoder must learn how to map $\boldsymbol{x}$ to $\boldsymbol{h}$ in a way that generalizes. For unusual $\boldsymbol{x}$ that do not resemble the training data, a learned, parametric encoder may fail to find an $\boldsymbol{h}$ that results in accurate reconstruction or a sparse code. For the vast majority of formulations of sparse coding models, where the inference problem is convex, the optimization procedure will always find the optimal code (unless degenerate cases such as replicated weight vectors occur). Obviously, the sparsity and reconstruction costs can still rise on unfamiliar points, but this is due to generalization error in the decoder weights, rather than generalization error in the encoder. The lack of generalization error in sparse coding's optimization-based encoding process may result in better generalization when sparse coding is used as a feature extractor for a classifier than when a parametric function is used to predict the code. Coates and Ng (2011) demonstrated that sparse coding features generalize better for object recognition tasks than the features of a related model based on a parametric encoder, the linear-sigmoid autoencoder. Inspired by their work, Goodfellow *et al.* (2013d) showed that a variant of sparse coding generalizes better than other feature extractors in the regime where extremely few labels are available (twenty or fewer labels per class).

The primary disadvantage of the non-parametric encoder is that it requires greater time to compute $\boldsymbol{h}$ given $\boldsymbol{x}$ because the non-parametric approach requires running an iterative algorithm. The parametric autoencoder approach, developed in chapter 14, uses only a fixed number of layers, often only one. Another disadvantage is that it is not straight-forward to back-propagate through the non-parametric encoder, which makes it difficult to pretrain a sparse coding model with an unsupervised criterion and then fine-tune it using a supervised criterion. Modified versions of sparse coding that permit approximate derivatives do exist but are not widely used (Bagnell and Bradley, 2009).

Sparse coding, like other linear factor models, often produces poor samples, as shown in figure 13.2. This happens even when the model is able to reconstruct the data well and provide useful features for a classifier. The reason is that each individual feature may be learned well, but the factorial prior on the hidden code results in the model including random subsets of all of the features in each generated sample. This motivates the development of deeper models that can impose a non-

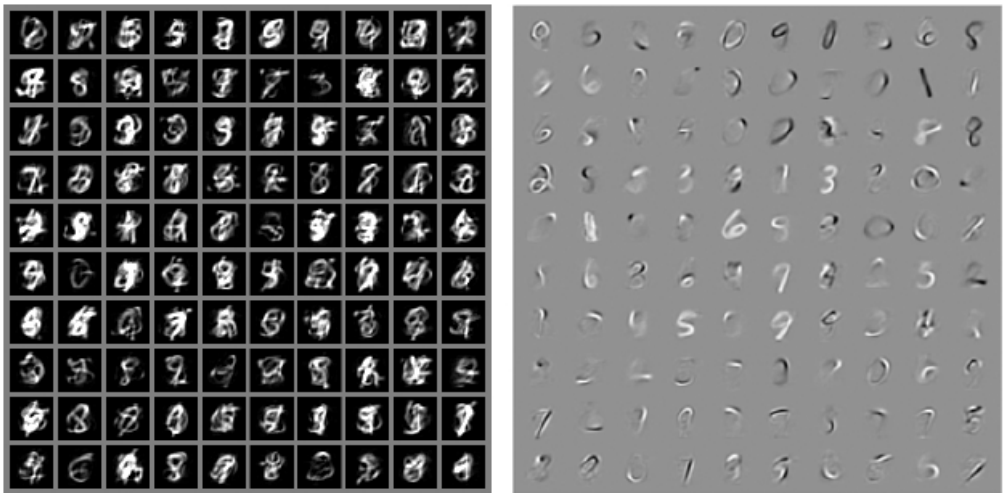


Figure 13.2: Example samples and weights from a spike and slab sparse coding model trained on the MNIST dataset. *(Left)*The samples from the model do not resemble the training examples. At first glance, one might assume the model is poorly fit. *(Right)*The weight vectors of the model have learned to represent penstrokes and sometimes complete digits. The model has thus learned useful features. The problem is that the factorial prior over features results in random subsets of features being combined. Few such subsets are appropriate to form a recognizable MNIST digit. This motivates the development of generative models that have more powerful distributions over their latent codes. Figure reproduced with permission from Goodfellow *et al.* (2013d).

factorial distribution on the deepest code layer, as well as the development of more sophisticated shallow models.

13.5 Manifold Interpretation of PCA

Linear factor models including PCA and factor analysis can be interpreted as learning a manifold (Hinton *et al.*, 1997). We can view probabilistic PCA as defining a thin pancake-shaped region of high probability—a Gaussian distribution that is very narrow along some axes, just as a pancake is very flat along its vertical axis, but is elongated along other axes, just as a pancake is wide along its horizontal axes. This is illustrated in figure 13.3. PCA can be interpreted as aligning this pancake with a linear manifold in a higher-dimensional space. This interpretation applies not just to traditional PCA but also to any linear autoencoder that learns matrices $\boldsymbol{W}$ and $\boldsymbol{V}$ with the goal of making the reconstruction of $\boldsymbol{x}$ lie as close to $\boldsymbol{x}$ as possible,

Let the encoder be

$$\boldsymbol{h} = f(\boldsymbol{x}) = \boldsymbol{W}^\top(\boldsymbol{x} - \boldsymbol{\mu}). \tag{13.19}$$

The encoder computes a low-dimensional representation of h. With the autoencoder view, we have a decoder computing the reconstruction

$$\hat{\boldsymbol{x}} = g(\boldsymbol{h}) = \boldsymbol{b} + \boldsymbol{V}\boldsymbol{h}. \tag{13.20}$$

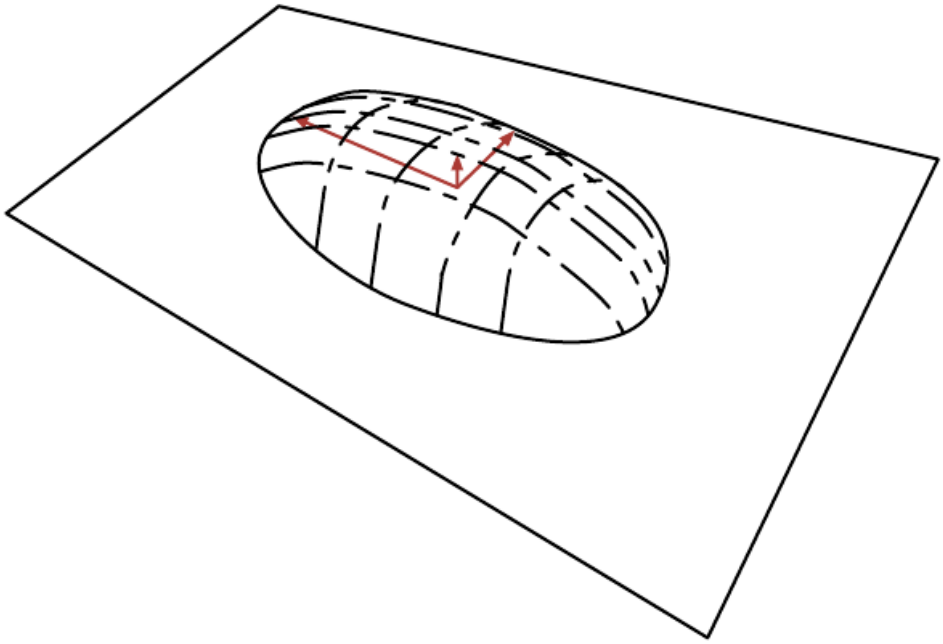


Figure 13.3: Flat Gaussian capturing probability concentration near a low-dimensional manifold. The figure shows the upper half of the "pancake" above the "manifold plane" which goes through its middle. The variance in the direction orthogonal to the manifold is very small (arrow pointing out of plane) and can be considered like "noise," while the other variances are large (arrows in the plane) and correspond to "signal," and a coordinate system for the reduced-dimension data.

The choices of linear encoder and decoder that minimize reconstruction error

$$\mathbb{E}[||\boldsymbol{x} - \hat{\boldsymbol{x}}||^2] \tag{13.21}$$

correspond to $\boldsymbol{V} = \boldsymbol{W}$, $\boldsymbol{\mu} = \boldsymbol{b} = \mathbb{E}[\boldsymbol{x}]$ and the columns of $\boldsymbol{W}$ form an orthonormal basis which spans the same subspace as the principal eigenvectors of the covariance matrix

$$\boldsymbol{C} = \mathbb{E}[(\boldsymbol{x} - \boldsymbol{\mu})(\boldsymbol{x} - \boldsymbol{\mu})^\top]. \tag{13.22}$$

In the case of PCA, the columns of $\boldsymbol{W}$ are these eigenvectors, ordered by the magnitude of the corresponding eigenvalues (which are all real and non-negative).

One can also show that eigenvalue λ_i of $\boldsymbol{C}$ corresponds to the variance of $\boldsymbol{x}$ in the direction of eigenvector $\boldsymbol{v}^{(i)}$. If $\boldsymbol{x} \in \mathbb{R}^D$ and $\boldsymbol{h} \in \mathbb{R}^d$ with $d < D$, then the

optimal reconstruction error (choosing $\boldsymbol{\mu}$, $\boldsymbol{b}$, $\boldsymbol{V}$ and $\boldsymbol{W}$ as above) is

$$\min \mathbb{E}[||\boldsymbol{x} - \hat{\boldsymbol{x}}||^2] = \sum_{i=d+1}^{D} \lambda_i \,. \tag{13.23}$$

Hence, if the covariance has rank d, the eigenvalues λ_{d+1} to λ_D are 0 and reconstruction error is 0.

Furthermore, one can also show that the above solution can be obtained by maximizing the variances of the elements of $\boldsymbol{h}$, under orthogonal $\boldsymbol{W}$, instead of minimizing reconstruction error.

Linear factor models are some of the simplest generative models and some of the simplest models that learn a representation of data. Much as linear classifiers and linear regression models may be extended to deep feedforward networks, these linear factor models may be extended to autoencoder networks and deep probabilistic models that perform the same tasks but with a much more powerful and flexible model family.

Chapter 14

Autoencoders

An **autoencoder** is a neural network that is trained to attempt to copy its input to its output. Internally, it has a hidden layer $\boldsymbol{h}$ that describes a **code** used to represent the input. The network may be viewed as consisting of two parts: an encoder function $\boldsymbol{h} = f(\boldsymbol{x})$ and a decoder that produces a reconstruction $\boldsymbol{r} = g(\boldsymbol{h})$. This architecture is presented in figure 14.1. If an autoencoder succeeds in simply learning to set $g(f(\boldsymbol{x})) = \boldsymbol{x}$ everywhere, then it is not especially useful. Instead, autoencoders are designed to be unable to learn to copy perfectly. Usually they are restricted in ways that allow them to copy only approximately, and to copy only input that resembles the training data. Because the model is forced to prioritize which aspects of the input should be copied, it often learns useful properties of the data.

Modern autoencoders have generalized the idea of an encoder and a decoder beyond deterministic functions to stochastic mappings $p_{\text{encoder}}(\boldsymbol{h} \mid \boldsymbol{x})$ and $p_{\text{decoder}}(\boldsymbol{x} \mid \boldsymbol{h})$.

The idea of autoencoders has been part of the historical landscape of neural networks for decades (LeCun, 1987; Bourlard and Kamp, 1988; Hinton and Zemel, 1994). Traditionally, autoencoders were used for dimensionality reduction or feature learning. Recently, theoretical connections between autoencoders and latent variable models have brought autoencoders to the forefront of generative modeling, as we will see in chapter 20. Autoencoders may be thought of as being a special case of feedforward networks, and may be trained with all of the same techniques, typically minibatch gradient descent following gradients computed by back-propagation. Unlike general feedforward networks, autoencoders may also be trained using **recirculation** (Hinton and McClelland, 1988), a learning algorithm based on comparing the activations of the network on the original input

to the activations on the reconstructed input. Recirculation is regarded as more biologically plausible than back-propagation, but is rarely used for machine learning applications.

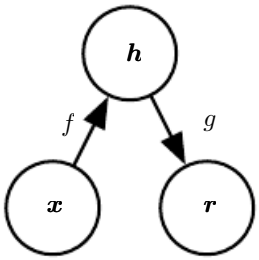


Figure 14.1: The general structure of an autoencoder, mapping an input $\boldsymbol{x}$ to an output (called reconstruction) $\boldsymbol{r}$ through an internal representation or code $\boldsymbol{h}$. The autoencoder has two components: the encoder f (mapping $\boldsymbol{x}$ to $\boldsymbol{h}$) and the decoder g (mapping $\boldsymbol{h}$ to $\boldsymbol{r}$).

14.1 Undercomplete Autoencoders

Copying the input to the output may sound useless, but we are typically not interested in the output of the decoder. Instead, we hope that training the autoencoder to perform the input copying task will result in $\boldsymbol{h}$ taking on useful properties.

One way to obtain useful features from the autoencoder is to constrain $\boldsymbol{h}$ to have smaller dimension than $\boldsymbol{x}$. An autoencoder whose code dimension is less than the input dimension is called **undercomplete**. Learning an undercomplete representation forces the autoencoder to capture the most salient features of the training data.

The learning process is described simply as minimizing a loss function

$$L(\boldsymbol{x}, g(f(\boldsymbol{x}))) \tag{14.1}$$

where L is a loss function penalizing $g(f(\boldsymbol{x}))$ for being dissimilar from $\boldsymbol{x}$, such as the mean squared error.

When the decoder is linear and L is the mean squared error, an undercomplete autoencoder learns to span the same subspace as PCA. In this case, an autoencoder trained to perform the copying task has learned the principal subspace of the training data as a side-effect.

Autoencoders with nonlinear encoder functions f and nonlinear decoder functions g can thus learn a more powerful nonlinear generalization of PCA. Unfortu-

nately, if the encoder and decoder are allowed too much capacity, the autoencoder can learn to perform the copying task without extracting useful information about the distribution of the data. Theoretically, one could imagine that an autoencoder with a one-dimensional code but a very powerful nonlinear encoder could learn to represent each training example $\boldsymbol{x}^{(i)}$ with the code i. The decoder could learn to map these integer indices back to the values of specific training examples. This specific scenario does not occur in practice, but it illustrates clearly that an autoencoder trained to perform the copying task can fail to learn anything useful about the dataset if the capacity of the autoencoder is allowed to become too great.

14.2 Regularized Autoencoders

Undercomplete autoencoders, with code dimension less than the input dimension, can learn the most salient features of the data distribution. We have seen that these autoencoders fail to learn anything useful if the encoder and decoder are given too much capacity.

A similar problem occurs if the hidden code is allowed to have dimension equal to the input, and in the **overcomplete** case in which the hidden code has dimension greater than the input. In these cases, even a linear encoder and linear decoder can learn to copy the input to the output without learning anything useful about the data distribution.

Ideally, one could train any architecture of autoencoder successfully, choosing the code dimension and the capacity of the encoder and decoder based on the complexity of distribution to be modeled. Regularized autoencoders provide the ability to do so. Rather than limiting the model capacity by keeping the encoder and decoder shallow and the code size small, regularized autoencoders use a loss function that encourages the model to have other properties besides the ability to copy its input to its output. These other properties include sparsity of the representation, smallness of the derivative of the representation, and robustness to noise or to missing inputs. A regularized autoencoder can be nonlinear and overcomplete but still learn something useful about the data distribution even if the model capacity is great enough to learn a trivial identity function.

In addition to the methods described here which are most naturally interpreted as regularized autoencoders, nearly any generative model with latent variables and equipped with an inference procedure (for computing latent representations given input) may be viewed as a particular form of autoencoder. Two generative modeling approaches that emphasize this connection with autoencoders are the descendants of the Helmholtz machine (Hinton *et al.*, 1995b), such as the variational

autoencoder (section 20.10.3) and the generative stochastic networks (section 20.12). These models naturally learn high-capacity, overcomplete encodings of the input and do not require regularization for these encodings to be useful. Their encodings are naturally useful because the models were trained to approximately maximize the probability of the training data rather than to copy the input to the output.

14.2.1 Sparse Autoencoders

A sparse autoencoder is simply an autoencoder whose training criterion involves a sparsity penalty $\Omega(\boldsymbol{h})$ on the code layer $\boldsymbol{h}$, in addition to the reconstruction error:

$$L(\boldsymbol{x}, g(f(\boldsymbol{x}))) + \Omega(\boldsymbol{h}) \tag{14.2}$$

where $g(\boldsymbol{h})$ is the decoder output and typically we have $\boldsymbol{h} = f(\boldsymbol{x})$, the encoder output.

Sparse autoencoders are typically used to learn features for another task such as classification. An autoencoder that has been regularized to be sparse must respond to unique statistical features of the dataset it has been trained on, rather than simply acting as an identity function. In this way, training to perform the copying task with a sparsity penalty can yield a model that has learned useful features as a byproduct.

We can think of the penalty $\Omega(\boldsymbol{h})$ simply as a regularizer term added to a feedforward network whose primary task is to copy the input to the output (unsupervised learning objective) and possibly also perform some supervised task (with a supervised learning objective) that depends on these sparse features. Unlike other regularizers such as weight decay, there is not a straightforward Bayesian interpretation to this regularizer. As described in section 5.6.1, training with weight decay and other regularization penalties can be interpreted as a MAP approximation to Bayesian inference, with the added regularizing penalty corresponding to a prior probability distribution over the model parameters. In this view, regularized maximum likelihood corresponds to maximizing $p(\boldsymbol{\theta} \mid \boldsymbol{x})$, which is equivalent to maximizing $\log p(\boldsymbol{x} \mid \boldsymbol{\theta}) + \log p(\boldsymbol{\theta})$. The $\log p(\boldsymbol{x} \mid \boldsymbol{\theta})$ term is the usual data log-likelihood term and the $\log p(\boldsymbol{\theta})$ term, the log-prior over parameters, incorporates the preference over particular values of $\boldsymbol{\theta}$. This view was described in section 5.6. Regularized autoencoders defy such an interpretation because the regularizer depends on the data and is therefore by definition not a prior in the formal sense of the word. We can still think of these regularization terms as implicitly expressing a preference over functions.

Rather than thinking of the sparsity penalty as a regularizer for the copying task, we can think of the entire sparse autoencoder framework as approximating

maximum likelihood training of a generative model that has latent variables. Suppose we have a model with visible variables $\boldsymbol{x}$ and latent variables $\boldsymbol{h}$, with an explicit joint distribution $p_{\text{model}}(\boldsymbol{x}, \boldsymbol{h}) = p_{\text{model}}(\boldsymbol{h}) p_{\text{model}}(\boldsymbol{x} \mid \boldsymbol{h})$. We refer to $p_{\text{model}}(\boldsymbol{h})$ as the model's prior distribution over the latent variables, representing the model's beliefs prior to seeing $\boldsymbol{x}$. This is different from the way we have previously used the word "prior," to refer to the distribution $p(\boldsymbol{\theta})$ encoding our beliefs about the model's parameters before we have seen the training data. The log-likelihood can be decomposed as

$$\log p_{\text{model}}(\boldsymbol{x}) = \log \sum_{\boldsymbol{h}} p_{\text{model}}(\boldsymbol{h}, \boldsymbol{x}). \tag{14.3}$$

We can think of the autoencoder as approximating this sum with a point estimate for just one highly likely value for $\boldsymbol{h}$. This is similar to the sparse coding generative model (section 13.4), but with $\boldsymbol{h}$ being the output of the parametric encoder rather than the result of an optimization that infers the most likely $\boldsymbol{h}$. From this point of view, with this chosen $\boldsymbol{h}$, we are maximizing

$$\log p_{\text{model}}(\boldsymbol{h}, \boldsymbol{x}) = \log p_{\text{model}}(\boldsymbol{h}) + \log p_{\text{model}}(\boldsymbol{x} \mid \boldsymbol{h}). \tag{14.4}$$

The $\log p_{\text{model}}(\boldsymbol{h})$ term can be sparsity-inducing. For example, the Laplace prior,

$$p_{\text{model}}(h_i) = \frac{\lambda}{2} e^{-\lambda |h_i|}, \tag{14.5}$$

corresponds to an absolute value sparsity penalty. Expressing the log-prior as an absolute value penalty, we obtain

$$\Omega(\boldsymbol{h}) = \lambda \sum_i |h_i| \tag{14.6}$$

$$-\log p_{\text{model}}(\boldsymbol{h}) = \sum_i \left(\lambda |h_i| - \log \frac{\lambda}{2} \right) = \Omega(\boldsymbol{h}) + \text{const} \tag{14.7}$$

where the constant term depends only on λ and not $\boldsymbol{h}$. We typically treat λ as a hyperparameter and discard the constant term since it does not affect the parameter learning. Other priors such as the Student-t prior can also induce sparsity. From this point of view of sparsity as resulting from the effect of $p_{\text{model}}(\boldsymbol{h})$ on approximate maximum likelihood learning, the sparsity penalty is not a regularization term at all. It is just a consequence of the model's distribution over its latent variables. This view provides a different motivation for training an autoencoder: it is a way of approximately training a generative model. It also provides a different reason for

why the features learned by the autoencoder are useful: they describe the latent variables that explain the input.

Early work on sparse autoencoders (Ranzato *et al.*, 2007a, 2008) explored various forms of sparsity and proposed a connection between the sparsity penalty and the $\log Z$ term that arises when applying maximum likelihood to an undirected probabilistic model $p(\boldsymbol{x}) = \frac{1}{Z}\tilde{p}(\boldsymbol{x})$. The idea is that minimizing $\log Z$ prevents a probabilistic model from having high probability everywhere, and imposing sparsity on an autoencoder prevents the autoencoder from having low reconstruction error everywhere. In this case, the connection is on the level of an intuitive understanding of a general mechanism rather than a mathematical correspondence. The interpretation of the sparsity penalty as corresponding to $\log p_{\text{model}}(\boldsymbol{h})$ in a directed model $p_{\text{model}}(\boldsymbol{h})p_{\text{model}}(\boldsymbol{x} \mid \boldsymbol{h})$ is more mathematically straightforward.

One way to achieve *actual zeros* in $\boldsymbol{h}$ for sparse (and denoising) autoencoders was introduced in Glorot *et al.* (2011b). The idea is to use rectified linear units to produce the code layer. With a prior that actually pushes the representations to zero (like the absolute value penalty), one can thus indirectly control the average number of zeros in the representation.

14.2.2 Denoising Autoencoders

Rather than adding a penalty Ω to the cost function, we can obtain an autoencoder that learns something useful by changing the reconstruction error term of the cost function.

Traditionally, autoencoders minimize some function

$$L(\boldsymbol{x}, g(f(\boldsymbol{x}))) \tag{14.8}$$

where L is a loss function penalizing $g(f(\boldsymbol{x}))$ for being dissimilar from $\boldsymbol{x}$, such as the L^2 norm of their difference. This encourages $g \circ f$ to learn to be merely an identity function if they have the capacity to do so.

A **denoising autoencoder** or DAE instead minimizes

$$L(\boldsymbol{x}, g(f(\tilde{\boldsymbol{x}}))), \tag{14.9}$$

where $\tilde{\boldsymbol{x}}$ is a copy of $\boldsymbol{x}$ that has been corrupted by some form of noise. Denoising autoencoders must therefore undo this corruption rather than simply copying their input.

Denoising training forces f and g to implicitly learn the structure of $p_{\text{data}}(\boldsymbol{x})$, as shown by Alain and Bengio (2013) and Bengio *et al.* (2013c). Denoising

autoencoders thus provide yet another example of how useful properties can emerge as a byproduct of minimizing reconstruction error. They are also an example of how overcomplete, high-capacity models may be used as autoencoders so long as care is taken to prevent them from learning the identity function. Denoising autoencoders are presented in more detail in section 14.5.

14.2.3 Regularizing by Penalizing Derivatives

Another strategy for regularizing an autoencoder is to use a penalty Ω as in sparse autoencoders,

$$L(\boldsymbol{x}, g(f(\boldsymbol{x}))) + \Omega(\boldsymbol{h}, \boldsymbol{x}), \tag{14.10}$$

but with a different form of Ω:

$$\Omega(\boldsymbol{h}, \boldsymbol{x}) = \lambda \sum_i ||\nabla_{\boldsymbol{x}} h_i||^2. \tag{14.11}$$

This forces the model to learn a function that does not change much when $\boldsymbol{x}$ changes slightly. Because this penalty is applied only at training examples, it forces the autoencoder to learn features that capture information about the training distribution.

An autoencoder regularized in this way is called a **contractive autoencoder** or CAE. This approach has theoretical connections to denoising autoencoders, manifold learning and probabilistic modeling. The CAE is described in more detail in section 14.7.

14.3 Representational Power, Layer Size and Depth

Autoencoders are often trained with only a single layer encoder and a single layer decoder. However, this is not a requirement. In fact, using deep encoders and decoders offers many advantages.

Recall from section 6.4.1 that there are many advantages to depth in a feedforward network. Because autoencoders are feedforward networks, these advantages also apply to autoencoders. Moreover, the encoder is itself a feedforward network as is the decoder, so each of these components of the autoencoder can individually benefit from depth.

One major advantage of non-trivial depth is that the universal approximator theorem guarantees that a feedforward neural network with at least one hidden layer can represent an approximation of any function (within a broad class) to an

arbitrary degree of accuracy, provided that it has enough hidden units. This means that an autoencoder with a single hidden layer is able to represent the identity function along the domain of the data arbitrarily well. However, the mapping from input to code is shallow. This means that we are not able to enforce arbitrary constraints, such as that the code should be sparse. A deep autoencoder, with at least one additional hidden layer inside the encoder itself, can approximate any mapping from input to code arbitrarily well, given enough hidden units.

Depth can exponentially reduce the computational cost of representing some functions. Depth can also exponentially decrease the amount of training data needed to learn some functions. See section 6.4.1 for a review of the advantages of depth in feedforward networks.

Experimentally, deep autoencoders yield much better compression than corresponding shallow or linear autoencoders (Hinton and Salakhutdinov, 2006).

A common strategy for training a deep autoencoder is to greedily pretrain the deep architecture by training a stack of shallow autoencoders, so we often encounter shallow autoencoders, even when the ultimate goal is to train a deep autoencoder.

14.4 Stochastic Encoders and Decoders

Autoencoders are just feedforward networks. The same loss functions and output unit types that can be used for traditional feedforward networks are also used for autoencoders.

As described in section 6.2.2.4, a general strategy for designing the output units and the loss function of a feedforward network is to define an output distribution $p(\boldsymbol{y} \mid \boldsymbol{x})$ and minimize the negative log-likelihood $-\log p(\boldsymbol{y} \mid \boldsymbol{x})$. In that setting, $\boldsymbol{y}$ was a vector of targets, such as class labels.

In the case of an autoencoder, $\boldsymbol{x}$ is now the target as well as the input. However, we can still apply the same machinery as before. Given a hidden code $\boldsymbol{h}$, we may think of the decoder as providing a conditional distribution $p_{\text{decoder}}(\boldsymbol{x} \mid \boldsymbol{h})$. We may then train the autoencoder by minimizing $-\log p_{\text{decoder}}(\boldsymbol{x} \mid \boldsymbol{h})$. The exact form of this loss function will change depending on the form of p_{decoder}. As with traditional feedforward networks, we usually use linear output units to parametrize the mean of a Gaussian distribution if $\boldsymbol{x}$ is real-valued. In that case, the negative log-likelihood yields a mean squared error criterion. Similarly, binary $\boldsymbol{x}$ values correspond to a Bernoulli distribution whose parameters are given by a sigmoid output unit, discrete $\boldsymbol{x}$ values correspond to a softmax distribution, and so on.

Typically, the output variables are treated as being conditionally independent given $\boldsymbol{h}$ so that this probability distribution is inexpensive to evaluate, but some techniques such as mixture density outputs allow tractable modeling of outputs with correlations.

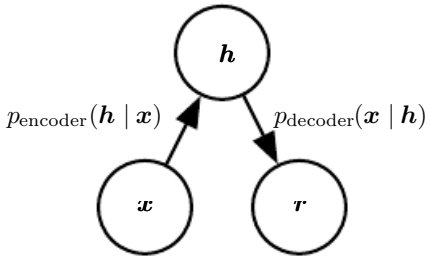


Figure 14.2: The structure of a stochastic autoencoder, in which both the encoder and the decoder are not simple functions but instead involve some noise injection, meaning that their output can be seen as sampled from a distribution, $p_{\text{encoder}}(\boldsymbol{h} \mid \boldsymbol{x})$ for the encoder and $p_{\text{decoder}}(\boldsymbol{x} \mid \boldsymbol{h})$ for the decoder.

To make a more radical departure from the feedforward networks we have seen previously, we can also generalize the notion of an **encoding function** $f(\boldsymbol{x})$ to an **encoding distribution** $p_{\text{encoder}}(\boldsymbol{h} \mid \boldsymbol{x})$, as illustrated in figure 14.2.

Any latent variable model $p_{\text{model}}(\boldsymbol{h}, \boldsymbol{x})$ defines a stochastic encoder

$$p_{\text{encoder}}(\boldsymbol{h} \mid \boldsymbol{x}) = p_{\text{model}}(\boldsymbol{h} \mid \boldsymbol{x}) \tag{14.12}$$

and a stochastic decoder

$$p_{\text{decoder}}(\boldsymbol{x} \mid \boldsymbol{h}) = p_{\text{model}}(\boldsymbol{x} \mid \boldsymbol{h}). \tag{14.13}$$

In general, the encoder and decoder distributions are not necessarily conditional distributions compatible with a unique joint distribution $p_{\text{model}}(\boldsymbol{x}, \boldsymbol{h})$. Alain *et al.* (2015) showed that training the encoder and decoder as a denoising autoencoder will tend to make them compatible asymptotically (with enough capacity and examples).

14.5 Denoising Autoencoders

The **denoising autoencoder** (DAE) is an autoencoder that receives a corrupted data point as input and is trained to predict the original, uncorrupted data point as its output.

The DAE training procedure is illustrated in figure 14.3. We introduce a corruption process $C(\tilde{\mathbf{x}} \mid \mathbf{x})$ which represents a conditional distribution over

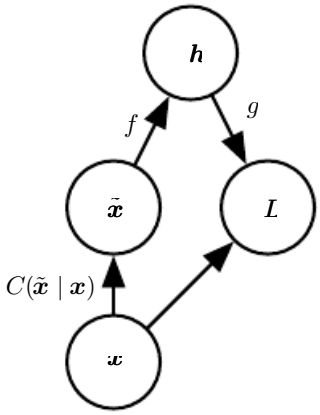


Figure 14.3: The computational graph of the cost function for a denoising autoencoder, which is trained to reconstruct the clean data point $\boldsymbol{x}$ from its corrupted version $\tilde{\boldsymbol{x}}$. This is accomplished by minimizing the loss $L = -\log p_{\text{decoder}}(\boldsymbol{x} \mid \boldsymbol{h} = f(\tilde{\boldsymbol{x}}))$, where $\tilde{\boldsymbol{x}}$ is a corrupted version of the data example $\boldsymbol{x}$, obtained through a given corruption process $C(\tilde{\boldsymbol{x}} \mid \boldsymbol{x})$. Typically the distribution p_{decoder} is a factorial distribution whose mean parameters are emitted by a feedforward network g.

corrupted samples $\tilde{\mathbf{x}}$, given a data sample $\mathbf{x}$. The autoencoder then learns a **reconstruction distribution** $p_{\text{reconstruct}}(\mathbf{x} \mid \tilde{\mathbf{x}})$ estimated from training pairs $(\boldsymbol{x}, \tilde{\boldsymbol{x}})$, as follows:

1. Sample a training example $\boldsymbol{x}$ from the training data.
2. Sample a corrupted version $\tilde{\boldsymbol{x}}$ from $C(\tilde{\mathbf{x}} \mid \mathbf{x} = \boldsymbol{x})$.
3. Use $(\boldsymbol{x}, \tilde{\boldsymbol{x}})$ as a training example for estimating the autoencoder reconstruction distribution $p_{\text{reconstruct}}(\boldsymbol{x} \mid \tilde{\boldsymbol{x}}) = p_{\text{decoder}}(\boldsymbol{x} \mid \boldsymbol{h})$ with $\boldsymbol{h}$ the output of encoder $f(\tilde{\boldsymbol{x}})$ and p_{decoder} typically defined by a decoder $g(\boldsymbol{h})$.

Typically we can simply perform gradient-based approximate minimization (such as minibatch gradient descent) on the negative log-likelihood $-\log p_{\text{decoder}}(\boldsymbol{x} \mid \boldsymbol{h})$. So long as the encoder is deterministic, the denoising autoencoder is a feedforward network and may be trained with exactly the same techniques as any other feedforward network.

We can therefore view the DAE as performing stochastic gradient descent on the following expectation:

$$-\mathbb{E}_{\mathbf{x} \sim \hat{p}_{\text{data}}(\mathbf{x})} \mathbb{E}_{\tilde{\mathbf{x}} \sim C(\tilde{\mathbf{x}} \mid \boldsymbol{x})} \log p_{\text{decoder}}(\boldsymbol{x} \mid \boldsymbol{h} = f(\tilde{\boldsymbol{x}})) \tag{14.14}$$

where $\hat{p}_{\text{data}}(\mathbf{x})$ is the training distribution.

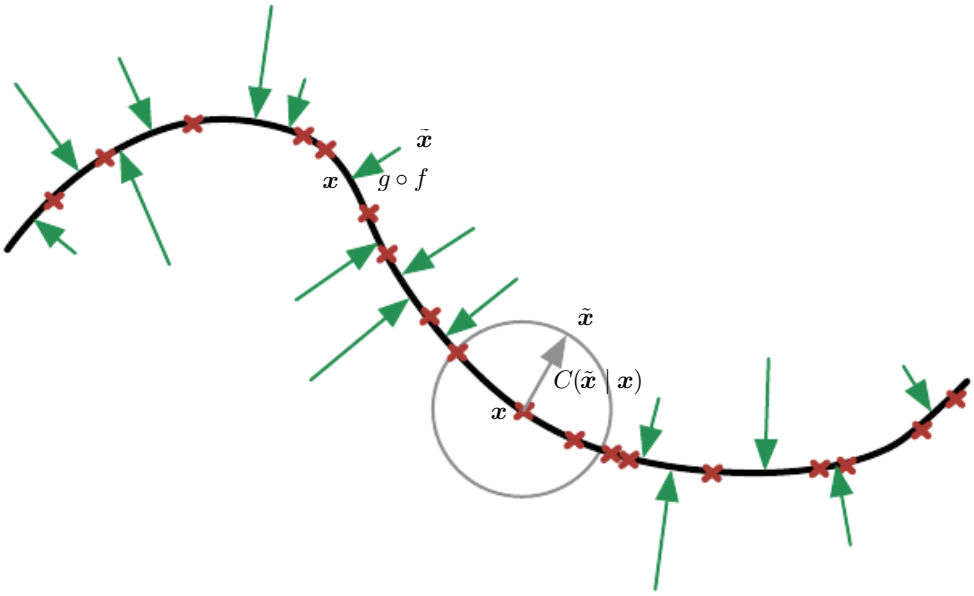


Figure 14.4: A denoising autoencoder is trained to map a corrupted data point $\tilde{\boldsymbol{x}}$ back to the original data point $\boldsymbol{x}$. We illustrate training examples $\boldsymbol{x}$ as red crosses lying near a low-dimensional manifold illustrated with the bold black line. We illustrate the corruption process $C(\tilde{\boldsymbol{x}} \mid \boldsymbol{x})$ with a gray circle of equiprobable corruptions. A gray arrow demonstrates how one training example is transformed into one sample from this corruption process. When the denoising autoencoder is trained to minimize the average of squared errors $||g(f(\tilde{\boldsymbol{x}})) - \boldsymbol{x}||^2$, the reconstruction $g(f(\tilde{\boldsymbol{x}}))$ estimates $\mathbb{E}_{\mathbf{x},\tilde{\mathbf{x}} \sim p_{\text{data}}(\mathbf{x}) C(\tilde{\mathbf{x}} \mid \mathbf{x})}[\mathbf{x} \mid \tilde{\boldsymbol{x}}]$. The vector $g(f(\tilde{\boldsymbol{x}})) - \tilde{\boldsymbol{x}}$ points approximately towards the nearest point on the manifold, since $g(f(\tilde{\boldsymbol{x}}))$ estimates the center of mass of the clean points $\boldsymbol{x}$ which could have given rise to $\tilde{\boldsymbol{x}}$. The autoencoder thus learns a vector field $g(f(\boldsymbol{x})) - \boldsymbol{x}$ indicated by the green arrows. This vector field estimates the score $\nabla_{\boldsymbol{x}} \log p_{\text{data}}(\boldsymbol{x})$ up to a multiplicative factor that is the average root mean square reconstruction error.

14.5.1 Estimating the Score

Score matching (Hyvärinen, 2005) is an alternative to maximum likelihood. It provides a consistent estimator of probability distributions based on encouraging the model to have the same **score** as the data distribution at every training point $\boldsymbol{x}$. In this context, the score is a particular gradient field:

$$\nabla_{\boldsymbol{x}} \log p(\boldsymbol{x}). \tag{14.15}$$

Score matching is discussed further in section 18.4. For the present discussion regarding autoencoders, it is sufficient to understand that learning the gradient field of $\log p_{\text{data}}$ is one way to learn the structure of p_{data} itself.

A very important property of DAEs is that their training criterion (with conditionally Gaussian $p(\boldsymbol{x} \mid \boldsymbol{h})$) makes the autoencoder learn a vector field $(g(f(\boldsymbol{x})) - \boldsymbol{x})$ that estimates the score of the data distribution. This is illustrated in figure 14.4.

Denoising training of a specific kind of autoencoder (sigmoidal hidden units, linear reconstruction units) using Gaussian noise and mean squared error as the reconstruction cost is equivalent (Vincent, 2011) to training a specific kind of undirected probabilistic model called an RBM with Gaussian visible units. This kind of model will be described in detail in section 20.5.1; for the present discussion it suffices to know that it is a model that provides an explicit $p_{\text{model}}(\boldsymbol{x}; \boldsymbol{\theta})$. When the RBM is trained using **denoising score matching** (Kingma and LeCun, 2010), its learning algorithm is equivalent to denoising training in the corresponding autoencoder. With a fixed noise level, regularized score matching is not a consistent estimator; it instead recovers a blurred version of the distribution. However, if the noise level is chosen to approach 0 when the number of examples approaches infinity, then consistency is recovered. Denoising score matching is discussed in more detail in section 18.5.

Other connections between autoencoders and RBMs exist. Score matching applied to RBMs yields a cost function that is identical to reconstruction error combined with a regularization term similar to the contractive penalty of the CAE (Swersky *et al.*, 2011). Bengio and Delalleau (2009) showed that an autoencoder gradient provides an approximation to contrastive divergence training of RBMs.

For continuous-valued $\boldsymbol{x}$, the denoising criterion with Gaussian corruption and reconstruction distribution yields an estimator of the score that is applicable to general encoder and decoder parametrizations (Alain and Bengio, 2013). This means a generic encoder-decoder architecture may be made to estimate the score

by training with the squared error criterion

$$||g(f(\tilde{\boldsymbol{x}})) - \boldsymbol{x}||^2 \tag{14.16}$$

and corruption

$$C(\tilde{\mathbf{x}} = \tilde{\boldsymbol{x}}|\boldsymbol{x}) = \mathcal{N}(\tilde{\boldsymbol{x}}; \mu = \boldsymbol{x}, \Sigma = \sigma^2 I) \tag{14.17}$$

with noise variance σ^2. See figure 14.5 for an illustration of how this works.

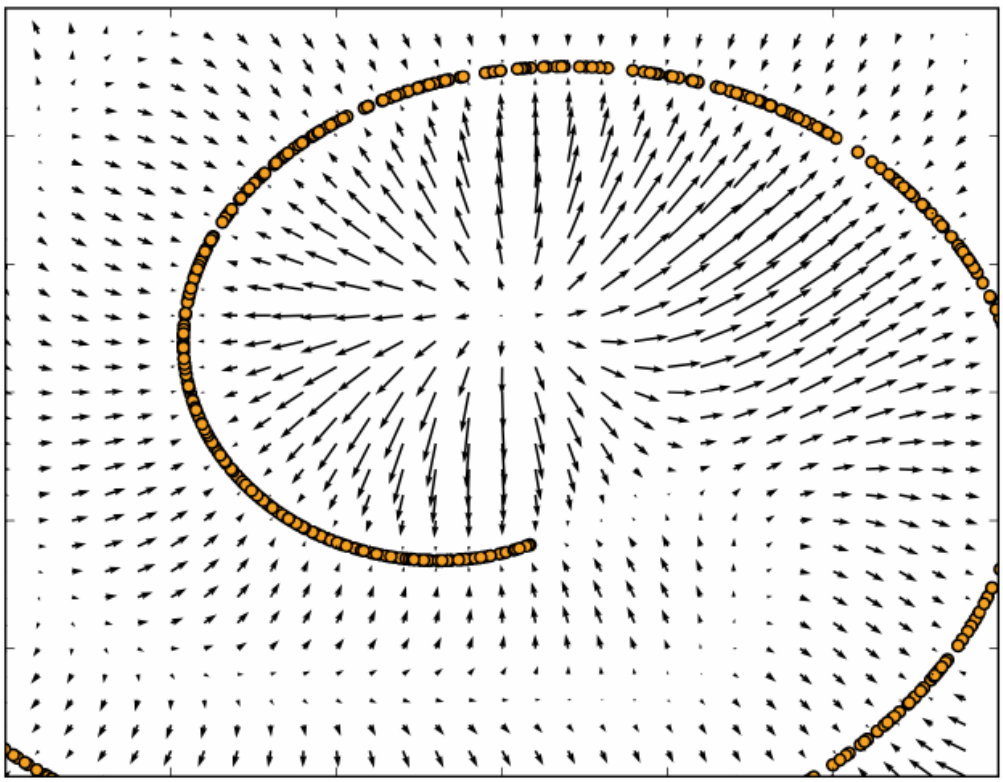


Figure 14.5: Vector field learned by a denoising autoencoder around a 1-D curved manifold near which the data concentrates in a 2-D space. Each arrow is proportional to the reconstruction minus input vector of the autoencoder and points towards higher probability according to the implicitly estimated probability distribution. The vector field has zeros at both maxima of the estimated density function (on the data manifolds) and at minima of that density function. For example, the spiral arm forms a one-dimensional manifold of local maxima that are connected to each other. Local minima appear near the middle of the gap between two arms. When the norm of reconstruction error (shown by the length of the arrows) is large, it means that probability can be significantly increased by moving in the direction of the arrow, and that is mostly the case in places of low probability. The autoencoder maps these low probability points to higher probability reconstructions. Where probability is maximal, the arrows shrink because the reconstruction becomes more accurate. Figure reproduced with permission from Alain and Bengio (2013).

In general, there is no guarantee that the reconstruction $g(f(\boldsymbol{x}))$ minus the input $\boldsymbol{x}$ corresponds to the gradient of any function, let alone to the score. That is

why the early results (Vincent, 2011) are specialized to particular parametrizations where $g(f(\boldsymbol{x})) - \boldsymbol{x}$ may be obtained by taking the derivative of another function. Kamyshanska and Memisevic (2015) generalized the results of Vincent (2011) by identifying a family of shallow autoencoders such that $g(f(\boldsymbol{x})) - \boldsymbol{x}$ corresponds to a score for all members of the family.

So far we have described only how the denoising autoencoder learns to represent a probability distribution. More generally, one may want to use the autoencoder as a generative model and draw samples from this distribution. This will be described later, in section 20.11.

14.5.1.1 Historical Perspective

The idea of using MLPs for denoising dates back to the work of LeCun (1987) and Gallinari *et al.* (1987). Behnke (2001) also used recurrent networks to denoise images. Denoising autoencoders are, in some sense, just MLPs trained to denoise. However, the name "denoising autoencoder" refers to a model that is intended not merely to learn to denoise its input but to learn a good internal representation as a side effect of learning to denoise. This idea came much later (Vincent *et al.*, 2008, 2010). The learned representation may then be used to pretrain a deeper unsupervised network or a supervised network. Like sparse autoencoders, sparse coding, contractive autoencoders and other regularized autoencoders, the motivation for DAEs was to allow the learning of a very high-capacity encoder while preventing the encoder and decoder from learning a useless identity function.

Prior to the introduction of the modern DAE, Inayoshi and Kurita (2005) explored some of the same goals with some of the same methods. Their approach minimizes reconstruction error in addition to a supervised objective while injecting noise in the hidden layer of a supervised MLP, with the objective to improve generalization by introducing the reconstruction error and the injected noise. However, their method was based on a linear encoder and could not learn function families as powerful as can the modern DAE.

14.6 Learning Manifolds with Autoencoders

Like many other machine learning algorithms, autoencoders exploit the idea that data concentrates around a low-dimensional manifold or a small set of such manifolds, as described in section 5.11.3. Some machine learning algorithms exploit this idea only insofar as that they learn a function that behaves correctly on the manifold but may have unusual behavior if given an input that is off the manifold.

Autoencoders take this idea further and aim to learn the structure of the manifold.

To understand how autoencoders do this, we must present some important characteristics of manifolds.

An important characterization of a manifold is the set of its **tangent planes**. At a point $\boldsymbol{x}$ on a d-dimensional manifold, the tangent plane is given by d basis vectors that span the local directions of variation allowed on the manifold. As illustrated in figure 14.6, these local directions specify how one can change $\boldsymbol{x}$ infinitesimally while staying on the manifold.

All autoencoder training procedures involve a compromise between two forces:

1. Learning a representation $\boldsymbol{h}$ of a training example $\boldsymbol{x}$ such that $\boldsymbol{x}$ can be approximately recovered from $\boldsymbol{h}$ through a decoder. The fact that $\boldsymbol{x}$ is drawn from the training data is crucial, because it means the autoencoder need not successfully reconstruct inputs that are not probable under the data generating distribution.

2. Satisfying the constraint or regularization penalty. This can be an architectural constraint that limits the capacity of the autoencoder, or it can be a regularization term added to the reconstruction cost. These techniques generally prefer solutions that are less sensitive to the input.

Clearly, neither force alone would be useful—copying the input to the output is not useful on its own, nor is ignoring the input. Instead, the two forces together are useful because they force the hidden representation to capture information about the structure of the data generating distribution. The important principle is that the autoencoder can afford to represent *only the variations that are needed to reconstruct training examples*. If the data generating distribution concentrates near a low-dimensional manifold, this yields representations that implicitly capture a local coordinate system for this manifold: only the variations tangent to the manifold around $\boldsymbol{x}$ need to correspond to changes in $\boldsymbol{h} = f(\boldsymbol{x})$. Hence the encoder learns a mapping from the input space $\boldsymbol{x}$ to a representation space, a mapping that is only sensitive to changes along the manifold directions, but that is insensitive to changes orthogonal to the manifold.

A one-dimensional example is illustrated in figure 14.7, showing that, by making the reconstruction function insensitive to perturbations of the input around the data points, we cause the autoencoder to recover the manifold structure.

To understand why autoencoders are useful for manifold learning, it is instructive to compare them to other approaches. What is most commonly learned to characterize a manifold is a **representation** of the data points on (or near)

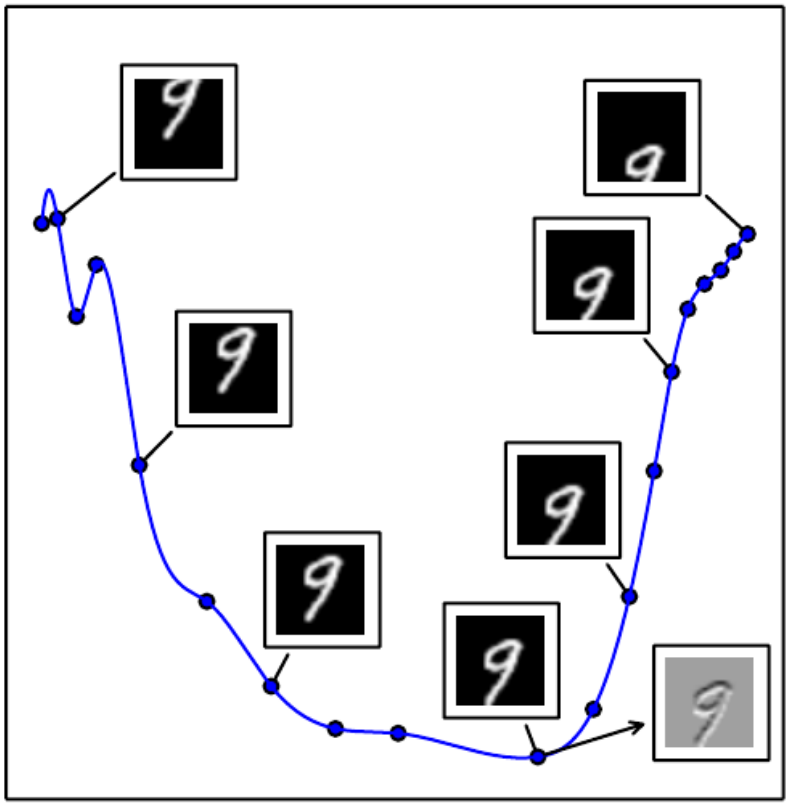


Figure 14.6: An illustration of the concept of a tangent hyperplane. Here we create a one-dimensional manifold in 784-dimensional space. We take an MNIST image with 784 pixels and transform it by translating it vertically. The amount of vertical translation defines a coordinate along a one-dimensional manifold that traces out a curved path through image space. This plot shows a few points along this manifold. For visualization, we have projected the manifold into two dimensional space using PCA. An n-dimensional manifold has an n-dimensional tangent plane at every point. This tangent plane touches the manifold exactly at that point and is oriented parallel to the surface at that point. It defines the space of directions in which it is possible to move while remaining on the manifold. This one-dimensional manifold has a single tangent line. We indicate an example tangent line at one point, with an image showing how this tangent direction appears in image space. Gray pixels indicate pixels that do not change as we move along the tangent line, white pixels indicate pixels that brighten, and black pixels indicate pixels that darken.

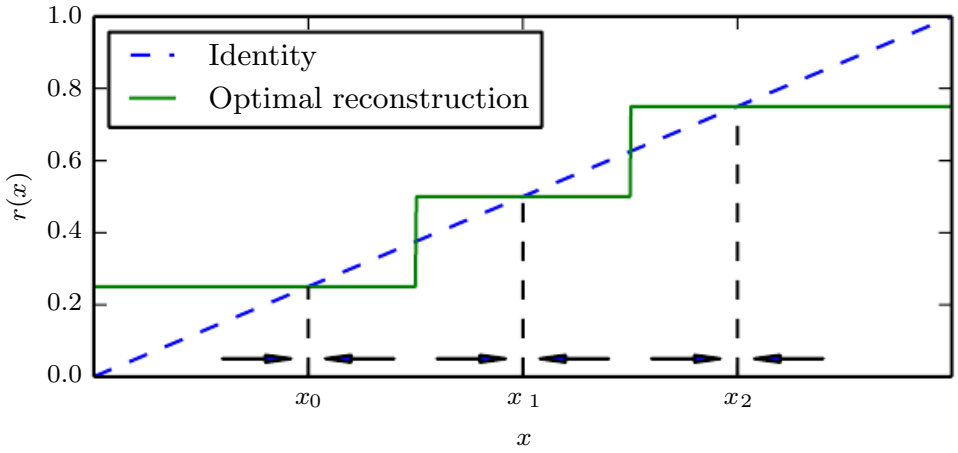


Figure 14.7: If the autoencoder learns a reconstruction function that is invariant to small perturbations near the data points, it captures the manifold structure of the data. Here the manifold structure is a collection of 0-dimensional manifolds. The dashed diagonal line indicates the identity function target for reconstruction. The optimal reconstruction function crosses the identity function wherever there is a data point. The horizontal arrows at the bottom of the plot indicate the $r(\boldsymbol{x}) - \boldsymbol{x}$ reconstruction direction vector at the base of the arrow, in input space, always pointing towards the nearest "manifold" (a single datapoint, in the 1-D case). The denoising autoencoder explicitly tries to make the derivative of the reconstruction function $r(\boldsymbol{x})$ small around the data points. The contractive autoencoder does the same for the encoder. Although the derivative of$r(\boldsymbol{x})$ is asked to be small around the data points, it can be large between the data points. The space between the data points corresponds to the region between the manifolds, where the reconstruction function must have a large derivative in order to map corrupted points back onto the manifold.

the manifold. Such a representation for a particular example is also called its embedding. It is typically given by a low-dimensional vector, with less dimensions than the "ambient" space of which the manifold is a low-dimensional subset. Some algorithms (non-parametric manifold learning algorithms, discussed below) directly learn an embedding for each training example, while others learn a more general mapping, sometimes called an encoder, or representation function, that maps any point in the ambient space (the input space) to its embedding.

Manifold learning has mostly focused on unsupervised learning procedures that attempt to capture these manifolds. Most of the initial machine learning research on learning nonlinear manifolds has focused on **non-parametric** methods based on the **nearest-neighbor graph**. This graph has one node per training example and edges connecting near neighbors to each other. These methods (Schölkopf *et al.*, 1998; Roweis and Saul, 2000; Tenenbaum *et al.*, 2000; Brand, 2003; Belkin

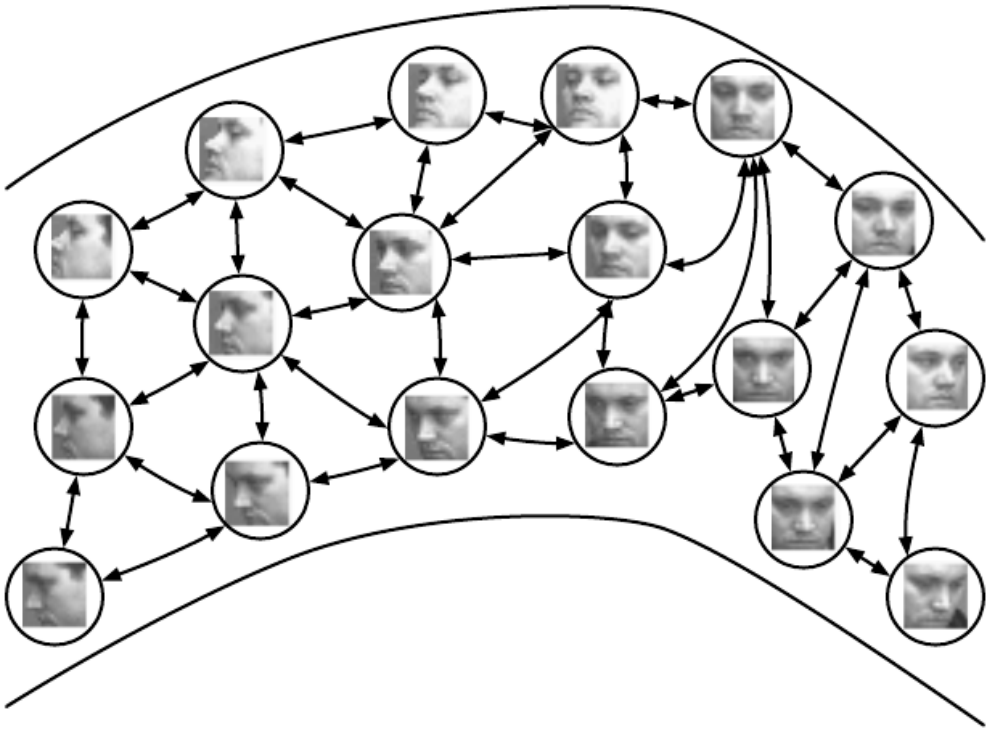


Figure 14.8: Non-parametric manifold learning procedures build a nearest neighbor graph in which nodes represent training examples a directed edges indicate nearest neighbor relationships. Various procedures can thus obtain the tangent plane associated with a neighborhood of the graph as well as a coordinate system that associates each training example with a real-valued vector position, or **embedding**. It is possible to generalize such a representation to new examples by a form of interpolation. So long as the number of examples is large enough to cover the curvature and twists of the manifold, these approaches work well. Images from the QMUL Multiview Face Dataset (Gong *et al.*, 2000).

and Niyogi, 2003; Donoho and Grimes, 2003; Weinberger and Saul, 2004; Hinton and Roweis, 2003; van der Maaten and Hinton, 2008) associate each of nodes with a tangent plane that spans the directions of variations associated with the difference vectors between the example and its neighbors, as illustrated in figure 14.8.

A global coordinate system can then be obtained through an optimization or solving a linear system. Figure 14.9 illustrates how a manifold can be tiled by a large number of locally linear Gaussian-like patches (or "pancakes," because the Gaussians are flat in the tangent directions).

However, there is a fundamental difficulty with such local non-parametric approaches to manifold learning, raised in Bengio and Monperrus (2005): if the manifolds are not very smooth (they have many peaks and troughs and twists), one may need a very large number of training examples to cover each one of

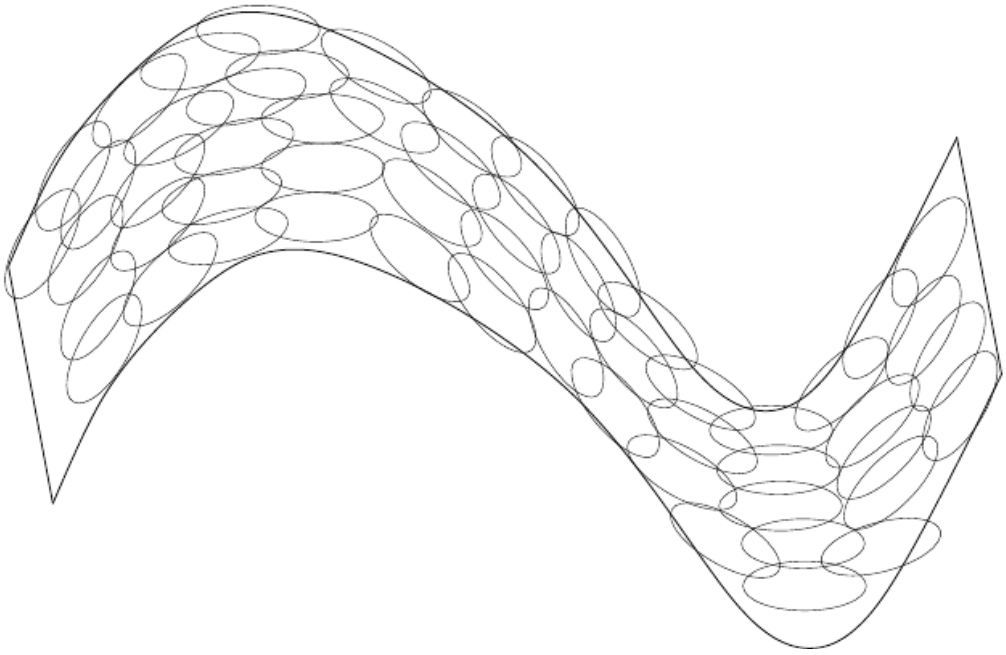


Figure 14.9: If the tangent planes (see figure 14.6) at each location are known, then they can be tiled to form a global coordinate system or a density function. Each local patch can be thought of as a local Euclidean coordinate system or as a locally flat Gaussian, or "pancake," with a very small variance in the directions orthogonal to the pancake and a very large variance in the directions defining the coordinate system on the pancake. A mixture of these Gaussians provides an estimated density function, as in the manifold Parzen window algorithm (Vincent and Bengio, 2003) or its non-local neural-net based variant (Bengio *et al.*, 2006c).

these variations, with no chance to generalize to unseen variations. Indeed, these methods can only generalize the shape of the manifold by interpolating between neighboring examples. Unfortunately, the manifolds involved in AI problems can have very complicated structure that can be difficult to capture from only local interpolation. Consider for example the manifold resulting from translation shown in figure 14.6. If we watch just one coordinate within the input vector, x_i, as the image is translated, we will observe that one coordinate encounters a peak or a trough in its value once for every peak or trough in brightness in the image. In other words, the complexity of the patterns of brightness in an underlying image template drives the complexity of the manifolds that are generated by performing simple image transformations. This motivates the use of distributed representations and deep learning for capturing manifold structure.

14.7 Contractive Autoencoders

The contractive autoencoder (Rifai *et al.*, 2011a,b) introduces an explicit regularizer on the code $\boldsymbol{h} = f(\boldsymbol{x})$, encouraging the derivatives of f to be as small as possible:

$$\Omega(\boldsymbol{h}) = \lambda \left\| \frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}} \right\|_F^2 . \tag{14.18}$$

The penalty $\Omega(\boldsymbol{h})$ is the squared Frobenius norm (sum of squared elements) of the Jacobian matrix of partial derivatives associated with the encoder function.

There is a connection between the denoising autoencoder and the contractive autoencoder: Alain and Bengio (2013) showed that in the limit of small Gaussian input noise, the denoising reconstruction error is equivalent to a contractive penalty on the reconstruction function that maps $\boldsymbol{x}$ to $\boldsymbol{r} = g(f(\boldsymbol{x}))$. In other words, denoising autoencoders make the reconstruction function resist small but finite-sized perturbations of the input, while contractive autoencoders make the feature extraction function resist infinitesimal perturbations of the input. When using the Jacobian-based contractive penalty to pretrain features $f(\boldsymbol{x})$ for use with a classifier, the best classification accuracy usually results from applying the contractive penalty to $f(\boldsymbol{x})$ rather than to $g(f(\boldsymbol{x}))$. A contractive penalty on $f(\boldsymbol{x})$ also has close connections to score matching, as discussed in section 14.5.1.

The name **contractive** arises from the way that the CAE warps space. Specifically, because the CAE is trained to resist perturbations of its input, it is encouraged to map a neighborhood of input points to a smaller neighborhood of output points. We can think of this as contracting the input neighborhood to a smaller output neighborhood.

To clarify, the CAE is contractive only locally—all perturbations of a training point $\boldsymbol{x}$ are mapped near to $f(\boldsymbol{x})$. Globally, two different points $\boldsymbol{x}$ and $\boldsymbol{x}'$ may be mapped to $f(\boldsymbol{x})$ and $f(\boldsymbol{x}')$ points that are farther apart than the original points. It is plausible that f be expanding in-between or far from the data manifolds (see for example what happens in the 1-D toy example of figure 14.7). When the $\Omega(\boldsymbol{h})$ penalty is applied to sigmoidal units, one easy way to shrink the Jacobian is to make the sigmoid units saturate to 0 or 1. This encourages the CAE to encode input points with extreme values of the sigmoid that may be interpreted as a binary code. It also ensures that the CAE will spread its code values throughout most of the hypercube that its sigmoidal hidden units can span.

We can think of the Jacobian matrix $\boldsymbol{J}$ at a point $\boldsymbol{x}$ as approximating the nonlinear encoder $f(\boldsymbol{x})$ as being a linear operator. This allows us to use the word "contractive" more formally. In the theory of linear operators, a linear operator

is said to be contractive if the norm of $\boldsymbol{J}\boldsymbol{x}$ remains less than or equal to 1 for all unit-norm $\boldsymbol{x}$. In other words, $\boldsymbol{J}$ is contractive if it shrinks the unit sphere. We can think of the CAE as penalizing the Frobenius norm of the local linear approximation of $f(\boldsymbol{x})$ at every training point $\boldsymbol{x}$ in order to encourage each of these local linear operator to become a contraction.

As described in section 14.6, regularized autoencoders learn manifolds by balancing two opposing forces. In the case of the CAE, these two forces are reconstruction error and the contractive penalty $\Omega(\boldsymbol{h})$. Reconstruction error alone would encourage the CAE to learn an identity function. The contractive penalty alone would encourage the CAE to learn features that are constant with respect to $\boldsymbol{x}$. The compromise between these two forces yields an autoencoder whose derivatives $\frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}}$ are mostly tiny. Only a small number of hidden units, corresponding to a small number of directions in the input, may have significant derivatives.

The goal of the CAE is to learn the manifold structure of the data. Directions $\boldsymbol{x}$ with large $\boldsymbol{J}\boldsymbol{x}$ rapidly change $\boldsymbol{h}$, so these are likely to be directions which approximate the tangent planes of the manifold. Experiments by Rifai *et al.* (2011a) and Rifai *et al.* (2011b) show that training the CAE results in most singular values of $\boldsymbol{J}$ dropping below 1 in magnitude and therefore becoming contractive. However, some singular values remain above 1, because the reconstruction error penalty encourages the CAE to encode the directions with the most local variance. The directions corresponding to the largest singular values are interpreted as the tangent directions that the contractive autoencoder has learned. Ideally, these tangent directions should correspond to real variations in the data. For example, a CAE applied to images should learn tangent vectors that show how the image changes as objects in the image gradually change pose, as shown in figure 14.6. Visualizations of the experimentally obtained singular vectors do seem to correspond to meaningful transformations of the input image, as shown in figure 14.10.

One practical issue with the CAE regularization criterion is that although it is cheap to compute in the case of a single hidden layer autoencoder, it becomes much more expensive in the case of deeper autoencoders. The strategy followed by Rifai *et al.* (2011a) is to separately train a series of single-layer autoencoders, each trained to reconstruct the previous autoencoder's hidden layer. The composition of these autoencoders then forms a deep autoencoder. Because each layer was separately trained to be locally contractive, the deep autoencoder is contractive as well. The result is not the same as what would be obtained by jointly training the entire architecture with a penalty on the Jacobian of the deep model, but it captures many of the desirable qualitative characteristics.

Another practical issue is that the contraction penalty can obtain useless results

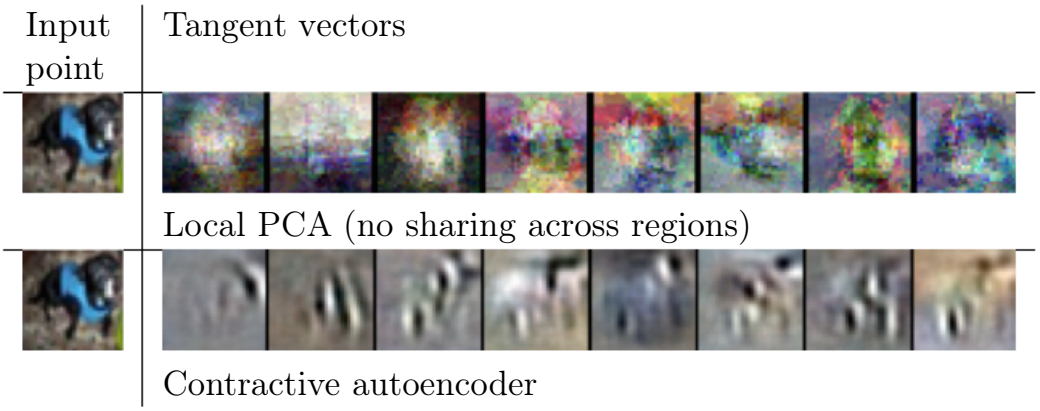


Figure 14.10: Illustration of tangent vectors of the manifold estimated by local PCA and by a contractive autoencoder. The location on the manifold is defined by the input image of a dog drawn from the CIFAR-10 dataset. The tangent vectors are estimated by the leading singular vectors of the Jacobian matrix $\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{x}}$ of the input-to-code mapping. Although both local PCA and the CAE can capture local tangents, the CAE is able to form more accurate estimates from limited training data because it exploits parameter sharing across different locations that share a subset of active hidden units. The CAE tangent directions typically correspond to moving or changing parts of the object (such as the head or legs). Images reproduced with permission from Rifai *et al.* (2011c).

if we do not impose some sort of scale on the decoder. For example, the encoder could consist of multiplying the input by a small constant ϵ and the decoder could consist of dividing the code by ϵ. As ϵ approaches 0, the encoder drives the contractive penalty $\Omega(\boldsymbol{h})$ to approach 0 without having learned anything about the distribution. Meanwhile, the decoder maintains perfect reconstruction. In Rifai *et al.* (2011a), this is prevented by tying the weights of f and g. Both f and g are standard neural network layers consisting of an affine transformation followed by an element-wise nonlinearity, so it is straightforward to set the weight matrix of g to be the transpose of the weight matrix of f.

14.8 Predictive Sparse Decomposition

Predictive sparse decomposition (PSD) is a model that is a hybrid of sparse coding and parametric autoencoders (Kavukcuoglu *et al.*, 2008). A parametric encoder is trained to predict the output of iterative inference. PSD has been applied to unsupervised feature learning for object recognition in images and video (Kavukcuoglu *et al.*, 2009, 2010; Jarrett *et al.*, 2009; Farabet *et al.*, 2011), as well as for audio (Henaff *et al.*, 2011). The model consists of an encoder $f(\boldsymbol{x})$ and a decoder $g(\boldsymbol{h})$ that are both parametric. During training, $\boldsymbol{h}$ is controlled by the

optimization algorithm. Training proceeds by minimizing

$$||\boldsymbol{x} - g(\boldsymbol{h})||^2 + \lambda|\boldsymbol{h}|_1 + \gamma||\boldsymbol{h} - f(\boldsymbol{x})||^2. \tag{14.19}$$

Like in sparse coding, the training algorithm alternates between minimization with respect to $\boldsymbol{h}$ and minimization with respect to the model parameters. Minimization with respect to $\boldsymbol{h}$ is fast because $f(\boldsymbol{x})$ provides a good initial value of $\boldsymbol{h}$ and the cost function constrains $\boldsymbol{h}$ to remain near $f(\boldsymbol{x})$ anyway. Simple gradient descent can obtain reasonable values of $\boldsymbol{h}$ in as few as ten steps.

The training procedure used by PSD is different from first training a sparse coding model and then training $f(\boldsymbol{x})$ to predict the values of the sparse coding features. The PSD training procedure regularizes the decoder to use parameters for which $f(\boldsymbol{x})$ can infer good code values.

Predictive sparse coding is an example of **learned approximate inference**. In section 19.5, this topic is developed further. The tools presented in chapter 19 make it clear that PSD can be interpreted as training a directed sparse coding probabilistic model by maximizing a lower bound on the log-likelihood of the model.

In practical applications of PSD, the iterative optimization is only used during training. The parametric encoder f is used to compute the learned features when the model is deployed. Evaluating f is computationally inexpensive compared to inferring $\boldsymbol{h}$ via gradient descent. Because f is a differentiable parametric function, PSD models may be stacked and used to initialize a deep network to be trained with another criterion.

14.9 Applications of Autoencoders

Autoencoders have been successfully applied to dimensionality reduction and information retrieval tasks. Dimensionality reduction was one of the first applications of representation learning and deep learning. It was one of the early motivations for studying autoencoders. For example, Hinton and Salakhutdinov (2006) trained a stack of RBMs and then used their weights to initialize a deep autoencoder with gradually smaller hidden layers, culminating in a bottleneck of 30 units. The resulting code yielded less reconstruction error than PCA into 30 dimensions and the learned representation was qualitatively easier to interpret and relate to the underlying categories, with these categories manifesting as well-separated clusters.

Lower-dimensional representations can improve performance on many tasks, such as classification. Models of smaller spaces consume less memory and runtime.

Many forms of dimensionality reduction place semantically related examples near each other, as observed by Salakhutdinov and Hinton (2007b) and Torralba *et al.* (2008). The hints provided by the mapping to the lower-dimensional space aid generalization.

One task that benefits even more than usual from dimensionality reduction is **information retrieval**, the task of finding entries in a database that resemble a query entry. This task derives the usual benefits from dimensionality reduction that other tasks do, but also derives the additional benefit that search can become extremely efficient in certain kinds of low dimensional spaces. Specifically, if we train the dimensionality reduction algorithm to produce a code that is low-dimensional and *binary*, then we can store all database entries in a hash table mapping binary code vectors to entries. This hash table allows us to perform information retrieval by returning all database entries that have the same binary code as the query. We can also search over slightly less similar entries very efficiently, just by flipping individual bits from the encoding of the query. This approach to information retrieval via dimensionality reduction and binarization is called **semantic hashing** (Salakhutdinov and Hinton, 2007b, 2009b), and has been applied to both textual input (Salakhutdinov and Hinton, 2007b, 2009b) and images (Torralba *et al.*, 2008; Weiss *et al.*, 2008; Krizhevsky and Hinton, 2011).

To produce binary codes for semantic hashing, one typically uses an encoding function with sigmoids on the final layer. The sigmoid units must be trained to be saturated to nearly 0 or nearly 1 for all input values. One trick that can accomplish this is simply to inject additive noise just before the sigmoid nonlinearity during training. The magnitude of the noise should increase over time. To fight that noise and preserve as much information as possible, the network must increase the magnitude of the inputs to the sigmoid function, until saturation occurs.

The idea of learning a hashing function has been further explored in several directions, including the idea of training the representations so as to optimize a loss more directly linked to the task of finding nearby examples in the hash table (Norouzi and Fleet, 2011).

Chapter 15

Representation Learning

In this chapter, we first discuss what it means to learn representations and how the notion of representation can be useful to design deep architectures. We discuss how learning algorithms share statistical strength across different tasks, including using information from unsupervised tasks to perform supervised tasks. Shared representations are useful to handle multiple modalities or domains, or to transfer learned knowledge to tasks for which few or no examples are given but a task representation exists. Finally, we step back and argue about the reasons for the success of representation learning, starting with the theoretical advantages of distributed representations (Hinton *et al.*, 1986) and deep representations and ending with the more general idea of underlying assumptions about the data generating process, in particular about underlying causes of the observed data.

Many information processing tasks can be very easy or very difficult depending on how the information is represented. This is a general principle applicable to daily life, computer science in general, and to machine learning. For example, it is straightforward for a person to divide 210 by 6 using long division. The task becomes considerably less straightforward if it is instead posed using the Roman numeral representation of the numbers. Most modern people asked to divide CCX by VI would begin by converting the numbers to the Arabic numeral representation, permitting long division procedures that make use of the place value system. More concretely, we can quantify the asymptotic runtime of various operations using appropriate or inappropriate representations. For example, inserting a number into the correct position in a sorted list of numbers is an $O(n)$ operation if the list is represented as a linked list, but only $O(\log n)$ if the list is represented as a red-black tree.

In the context of machine learning, what makes one representation better than

another? Generally speaking, a good representation is one that makes a subsequent learning task easier. The choice of representation will usually depend on the choice of the subsequent learning task.

We can think of feedforward networks trained by supervised learning as performing a kind of representation learning. Specifically, the last layer of the network is typically a linear classifier, such as a softmax regression classifier. The rest of the network learns to provide a representation to this classifier. Training with a supervised criterion naturally leads to the representation at every hidden layer (but more so near the top hidden layer) taking on properties that make the classification task easier. For example, classes that were not linearly separable in the input features may become linearly separable in the last hidden layer. In principle, the last layer could be another kind of model, such as a nearest neighbor classifier (Salakhutdinov and Hinton, 2007a). The features in the penultimate layer should learn different properties depending on the type of the last layer.

Supervised training of feedforward networks does not involve explicitly imposing any condition on the learned intermediate features. Other kinds of representation learning algorithms are often explicitly designed to shape the representation in some particular way. For example, suppose we want to learn a representation that makes density estimation easier. Distributions with more independences are easier to model, so we could design an objective function that encourages the elements of the representation vector $\boldsymbol{h}$ to be independent. Just like supervised networks, unsupervised deep learning algorithms have a main training objective but also learn a representation as a side effect. Regardless of how a representation was obtained, it can be used for another task. Alternatively, multiple tasks (some supervised, some unsupervised) can be learned together with some shared internal representation.

Most representation learning problems face a tradeoff between preserving as much information about the input as possible and attaining nice properties (such as independence).

Representation learning is particularly interesting because it provides one way to perform unsupervised and semi-supervised learning. We often have very large amounts of unlabeled training data and relatively little labeled training data. Training with supervised learning techniques on the labeled subset often results in severe overfitting. Semi-supervised learning offers the chance to resolve this overfitting problem by also learning from the unlabeled data. Specifically, we can learn good representations for the unlabeled data, and then use these representations to solve the supervised learning task.

Humans and animals are able to learn from very few labeled examples. We do

not yet know how this is possible. Many factors could explain improved human performance—for example, the brain may use very large ensembles of classifiers or Bayesian inference techniques. One popular hypothesis is that the brain is able to leverage unsupervised or semi-supervised learning. There are many ways to leverage unlabeled data. In this chapter, we focus on the hypothesis that the unlabeled data can be used to learn a good representation.

15.1 Greedy Layer-Wise Unsupervised Pretraining

Unsupervised learning played a key historical role in the revival of deep neural networks, enabling researchers for the first time to train a deep supervised network without requiring architectural specializations like convolution or recurrence. We call this procedure **unsupervised pretraining**, or more precisely, **greedy layer-wise unsupervised pretraining**. This procedure is a canonical example of how a representation learned for one task (unsupervised learning, trying to capture the shape of the input distribution) can sometimes be useful for another task (supervised learning with the same input domain).

Greedy layer-wise unsupervised pretraining relies on a single-layer representation learning algorithm such as an RBM, a single-layer autoencoder, a sparse coding model, or another model that learns latent representations. Each layer is pretrained using unsupervised learning, taking the output of the previous layer and producing as output a new representation of the data, whose distribution (or its relation to other variables such as categories to predict) is hopefully simpler. See algorithm 15.1 for a formal description.

Greedy layer-wise training procedures based on unsupervised criteria have long been used to sidestep the difficulty of jointly training the layers of a deep neural net for a supervised task. This approach dates back at least as far as the Neocognitron (Fukushima, 1975). The deep learning renaissance of 2006 began with the discovery that this greedy learning procedure could be used to find a good initialization for a joint learning procedure over all the layers, and that this approach could be used to successfully train even fully connected architectures (Hinton *et al.*, 2006; Hinton and Salakhutdinov, 2006; Hinton, 2006; Bengio *et al.*, 2007; Ranzato *et al.*, 2007a). Prior to this discovery, only convolutional deep networks or networks whose depth resulted from recurrence were regarded as feasible to train. Today, we now know that greedy layer-wise pretraining is not required to train fully connected deep architectures, but the unsupervised pretraining approach was the first method to succeed.

Greedy layer-wise pretraining is called **greedy** because it is a **greedy algo-**

rithm, meaning that it optimizes each piece of the solution independently, one piece at a time, rather than jointly optimizing all pieces. It is called **layer-wise** because these independent pieces are the layers of the network. Specifically, greedy layer-wise pretraining proceeds one layer at a time, training the k-th layer while keeping the previous ones fixed. In particular, the lower layers (which are trained first) are not adapted after the upper layers are introduced. It is called **unsupervised** because each layer is trained with an unsupervised representation learning algorithm. However it is also called **pretraining**, because it is supposed to be only a first step before a joint training algorithm is applied to **fine-tune** all the layers together. In the context of a supervised learning task, it can be viewed as a regularizer (in some experiments, pretraining decreases test error without decreasing training error) and a form of parameter initialization.

It is common to use the word "pretraining" to refer not only to the pretraining stage itself but to the entire two phase protocol that combines the pretraining phase and a supervised learning phase. The supervised learning phase may involve training a simple classifier on top of the features learned in the pretraining phase, or it may involve supervised fine-tuning of the entire network learned in the pretraining phase. No matter what kind of unsupervised learning algorithm or what model type is employed, in the vast majority of cases, the overall training scheme is nearly the same. While the choice of unsupervised learning algorithm will obviously impact the details, most applications of unsupervised pretraining follow this basic protocol.

Greedy layer-wise unsupervised pretraining can also be used as initialization for other unsupervised learning algorithms, such as deep autoencoders (Hinton and Salakhutdinov, 2006) and probabilistic models with many layers of latent variables. Such models include deep belief networks (Hinton *et al.*, 2006) and deep Boltzmann machines (Salakhutdinov and Hinton, 2009a). These deep generative models will be described in chapter 20.

As discussed in section 8.7.4, it is also possible to have greedy layer-wise *supervised* pretraining. This builds on the premise that training a shallow network is easier than training a deep one, which seems to have been validated in several contexts (Erhan *et al.*, 2010).

15.1.1 When and Why Does Unsupervised Pretraining Work?

On many tasks, greedy layer-wise unsupervised pretraining can yield substantial improvements in test error for classification tasks. This observation was responsible for the renewed interested in deep neural networks starting in 2006 (Hinton *et al.*,

Algorithm 15.1 *Greedy layer-wise unsupervised pretraining protocol.*
Given the following: Unsupervised feature learning algorithm $\mathcal{L}$, which takes a training set of examples and returns an encoder or feature function f. The raw input data is $\boldsymbol{X}$, with one row per example and $f^{(1)}(\boldsymbol{X})$ is the output of the first stage encoder on $\boldsymbol{X}$. In the case where fine-tuning is performed, we use a learner $\mathcal{T}$ which takes an initial function f, input examples $\boldsymbol{X}$ (and in the supervised fine-tuning case, associated targets $\boldsymbol{Y}$), and returns a tuned function. The number of stages is m.

$f \leftarrow$ Identity function
$\tilde{\boldsymbol{X}} = \boldsymbol{X}$
for $k = 1, \ldots, m$ **do**
  $f^{(k)} = \mathcal{L}(\tilde{\boldsymbol{X}})$
  $f \leftarrow f^{(k)} \circ f$
  $\tilde{\boldsymbol{X}} \leftarrow f^{(k)}(\tilde{\boldsymbol{X}})$
end for
if *fine-tuning* ***then***
  $f \leftarrow \mathcal{T}(f, \boldsymbol{X}, \boldsymbol{Y})$
end if
Return f

2006; Bengio *et al.*, 2007; Ranzato *et al.*, 2007a). On many other tasks, however, unsupervised pretraining either does not confer a benefit or even causes noticeable harm. Ma *et al.* (2015) studied the effect of pretraining on machine learning models for chemical activity prediction and found that, on average, pretraining was slightly harmful, but for many tasks was significantly helpful. Because unsupervised pretraining is sometimes helpful but often harmful it is important to understand when and why it works in order to determine whether it is applicable to a particular task.

At the outset, it is important to clarify that most of this discussion is restricted to greedy unsupervised pretraining in particular. There are other, completely different paradigms for performing semi-supervised learning with neural networks, such as virtual adversarial training described in section 7.13. It is also possible to train an autoencoder or generative model at the same time as the supervised model. Examples of this single-stage approach include the discriminative RBM (Larochelle and Bengio, 2008) and the ladder network (Rasmus *et al.*, 2015), in which the total objective is an explicit sum of the two terms (one using the labels and one only using the input).

Unsupervised pretraining combines two different ideas. First, it makes use of

the idea that the choice of initial parameters for a deep neural network can have a significant regularizing effect on the model (and, to a lesser extent, that it can improve optimization). Second, it makes use of the more general idea that learning about the input distribution can help to learn about the mapping from inputs to outputs.

Both of these ideas involve many complicated interactions between several parts of the machine learning algorithm that are not entirely understood.

The first idea, that the choice of initial parameters for a deep neural network can have a strong regularizing effect on its performance, is the least well understood. At the time that pretraining became popular, it was understood as initializing the model in a location that would cause it to approach one local minimum rather than another. Today, local minima are no longer considered to be a serious problem for neural network optimization. We now know that our standard neural network training procedures usually do not arrive at a critical point of any kind. It remains possible that pretraining initializes the model in a location that would otherwise be inaccessible—for example, a region that is surrounded by areas where the cost function varies so much from one example to another that minibatches give only a very noisy estimate of the gradient, or a region surrounded by areas where the Hessian matrix is so poorly conditioned that gradient descent methods must use very small steps. However, our ability to characterize exactly what aspects of the pretrained parameters are retained during the supervised training stage is limited. This is one reason that modern approaches typically use simultaneous unsupervised learning and supervised learning rather than two sequential stages. One may also avoid struggling with these complicated ideas about how optimization in the supervised learning stage preserves information from the unsupervised learning stage by simply freezing the parameters for the feature extractors and using supervised learning only to add a classifier on top of the learned features.

The other idea, that a learning algorithm can use information learned in the unsupervised phase to perform better in the supervised learning stage, is better understood. The basic idea is that some features that are useful for the unsupervised task may also be useful for the supervised learning task. For example, if we train a generative model of images of cars and motorcycles, it will need to know about wheels, and about how many wheels should be in an image. If we are fortunate, the representation of the wheels will take on a form that is easy for the supervised learner to access. This is not yet understood at a mathematical, theoretical level, so it is not always possible to predict which tasks will benefit from unsupervised learning in this way. Many aspects of this approach are highly dependent on the specific models used. For example, if we wish to add a linear classifier on

top of pretrained features, the features must make the underlying classes linearly separable. These properties often occur naturally but do not always do so. This is another reason that simultaneous supervised and unsupervised learning can be preferable—the constraints imposed by the output layer are naturally included from the start.

From the point of view of unsupervised pretraining as learning a representation, we can expect unsupervised pretraining to be more effective when the initial representation is poor. One key example of this is the use of word embeddings. Words represented by one-hot vectors are not very informative because every two distinct one-hot vectors are the same distance from each other (squared L^2 distance of 2). Learned word embeddings naturally encode similarity between words by their distance from each other. Because of this, unsupervised pretraining is especially useful when processing words. It is less useful when processing images, perhaps because images already lie in a rich vector space where distances provide a low quality similarity metric.

From the point of view of unsupervised pretraining as a regularizer, we can expect unsupervised pretraining to be most helpful when the number of labeled examples is very small. Because the source of information added by unsupervised pretraining is the unlabeled data, we may also expect unsupervised pretraining to perform best when the number of unlabeled examples is very large. The advantage of semi-supervised learning via unsupervised pretraining with many unlabeled examples and few labeled examples was made particularly clear in 2011 with unsupervised pretraining winning two international transfer learning competitions (Mesnil *et al.*, 2011; Goodfellow *et al.*, 2011), in settings where the number of labeled examples in the target task was small (from a handful to dozens of examples per class). These effects were also documented in carefully controlled experiments by Paine *et al.* (2014).

Other factors are likely to be involved. For example, unsupervised pretraining is likely to be most useful when the function to be learned is extremely complicated. Unsupervised learning differs from regularizers like weight decay because it does not bias the learner toward discovering a simple function but rather toward discovering feature functions that are useful for the unsupervised learning task. If the true underlying functions are complicated and shaped by regularities of the input distribution, unsupervised learning can be a more appropriate regularizer.

These caveats aside, we now analyze some success cases where unsupervised pretraining is known to cause an improvement, and explain what is known about why this improvement occurs. Unsupervised pretraining has usually been used to improve classifiers, and is usually most interesting from the point of view of

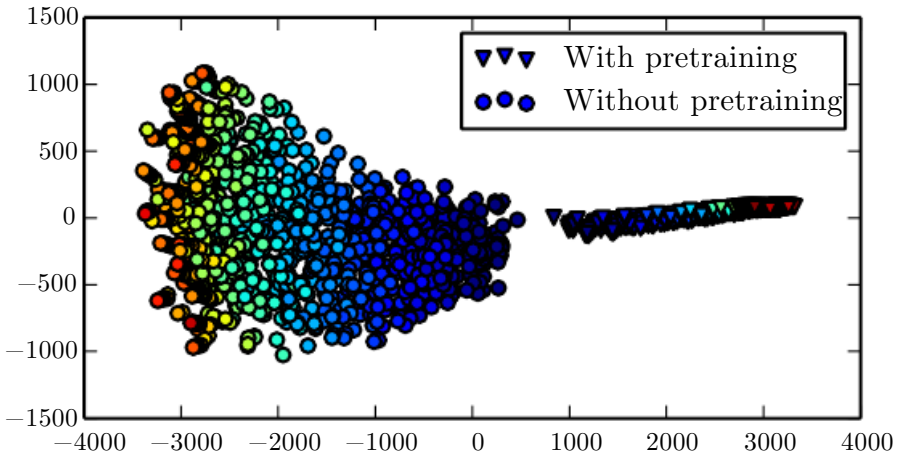


Figure 15.1: Visualization via nonlinear projection of the learning trajectories of different neural networks in *function space* (not parameter space, to avoid the issue of many-to-one mappings from parameter vectors to functions), with different random initializations and with or without unsupervised pretraining. Each point corresponds to a different neural network, at a particular time during its training process. This figure is adapted with permission from Erhan *et al.* (2010). A coordinate in function space is an infinite-dimensional vector associating every input $\boldsymbol{x}$ with an output $\boldsymbol{y}$. Erhan *et al.* (2010) made a linear projection to high-dimensional space by concatenating the $\boldsymbol{y}$ for many specific $\boldsymbol{x}$ points. They then made a further nonlinear projection to 2-D by Isomap (Tenenbaum *et al.*, 2000). Color indicates time. All networks are initialized near the center of the plot (corresponding to the region of functions that produce approximately uniform distributions over the class y for most inputs). Over time, learning moves the function outward, to points that make strong predictions. Training consistently terminates in one region when using pretraining and in another, non-overlapping region when not using pretraining. Isomap tries to preserve global relative distances (and hence volumes) so the small region corresponding to pretrained models may indicate that the pretraining-based estimator has reduced variance.

reducing test set error. However, unsupervised pretraining can help tasks other than classification, and can act to improve optimization rather than being merely a regularizer. For example, it can improve both train and test reconstruction error for deep autoencoders (Hinton and Salakhutdinov, 2006).

Erhan *et al.* (2010) performed many experiments to explain several successes of unsupervised pretraining. Both improvements to training error and improvements to test error may be explained in terms of unsupervised pretraining taking the parameters into a region that would otherwise be inaccessible. Neural network training is non-deterministic, and converges to a different function every time it is run. Training may halt at a point where the gradient becomes small, a point where early stopping ends training to prevent overfitting, or at a point where the gradient is large but it is difficult to find a downhill step due to problems such as stochasticity or poor conditioning of the Hessian. Neural networks that receive unsupervised pretraining consistently halt in the same region of function space, while neural networks without pretraining consistently halt in another region. See figure 15.1 for a visualization of this phenomenon. The region where pretrained networks arrive is smaller, suggesting that pretraining reduces the variance of the estimation process, which can in turn reduce the risk of severe over-fitting. In other words, unsupervised pretraining initializes neural network parameters into a region that they do not escape, and the results following this initialization are more consistent and less likely to be very bad than without this initialization.

Erhan *et al.* (2010) also provide some answers as to *when* pretraining works best—the mean and variance of the test error were most reduced by pretraining for deeper networks. Keep in mind that these experiments were performed before the invention and popularization of modern techniques for training very deep networks (rectified linear units, dropout and batch normalization) so less is known about the effect of unsupervised pretraining in conjunction with contemporary approaches.

An important question is how unsupervised pretraining can act as a regularizer. One hypothesis is that pretraining encourages the learning algorithm to discover features that relate to the underlying causes that generate the observed data. This is an important idea motivating many other algorithms besides unsupervised pretraining, and is described further in section 15.3.

Compared to other forms of unsupervised learning, unsupervised pretraining has the disadvantage that it operates with two separate training phases. Many regularization strategies have the advantage of allowing the user to control the strength of the regularization by adjusting the value of a single hyperparameter. Unsupervised pretraining does not offer a clear way to adjust the the strength of the regularization arising from the unsupervised stage. Instead, there are

very many hyperparameters, whose effect may be measured after the fact but is often difficult to predict ahead of time. When we perform unsupervised and supervised learning simultaneously, instead of using the pretraining strategy, there is a single hyperparameter, usually a coefficient attached to the unsupervised cost, that determines how strongly the unsupervised objective will regularize the supervised model. One can always predictably obtain less regularization by decreasing this coefficient. In the case of unsupervised pretraining, there is not a way of flexibly adapting the strength of the regularization—either the supervised model is initialized to pretrained parameters, or it is not.

Another disadvantage of having two separate training phases is that each phase has its own hyperparameters. The performance of the second phase usually cannot be predicted during the first phase, so there is a long delay between proposing hyperparameters for the first phase and being able to update them using feedback from the second phase. The most principled approach is to use validation set error in the supervised phase in order to select the hyperparameters of the pretraining phase, as discussed in Larochelle *et al.* (2009). In practice, some hyperparameters, like the number of pretraining iterations, are more conveniently set during the pretraining phase, using early stopping on the unsupervised objective, which is not ideal but computationally much cheaper than using the supervised objective.

Today, unsupervised pretraining has been largely abandoned, except in the field of natural language processing, where the natural representation of words as one-hot vectors conveys no similarity information and where very large unlabeled sets are available. In that case, the advantage of pretraining is that one can pretrain once on a huge unlabeled set (for example with a corpus containing billions of words), learn a good representation (typically of words, but also of sentences), and then use this representation or fine-tune it for a supervised task for which the training set contains substantially fewer examples. This approach was pioneered by by Collobert and Weston (2008b), Turian *et al.* (2010), and Collobert *et al.* (2011a) and remains in common use today.

Deep learning techniques based on supervised learning, regularized with dropout or batch normalization, are able to achieve human-level performance on very many tasks, but only with extremely large labeled datasets. These same techniques outperform unsupervised pretraining on medium-sized datasets such as CIFAR-10 and MNIST, which have roughly 5,000 labeled examples per class. On extremely small datasets, such as the alternative splicing dataset, Bayesian methods outperform methods based on unsupervised pretraining (Srivastava, 2013). For these reasons, the popularity of unsupervised pretraining has declined. Nevertheless, unsupervised pretraining remains an important milestone in the history of deep learning research

and continues to influence contemporary approaches. The idea of pretraining has been generalized to **supervised pretraining** discussed in section 8.7.4, as a very common approach for transfer learning. Supervised pretraining for transfer learning is popular (Oquab *et al.*, 2014; Yosinski *et al.*, 2014) for use with convolutional networks pretrained on the ImageNet dataset. Practitioners publish the parameters of these trained networks for this purpose, just like pretrained word vectors are published for natural language tasks (Collobert *et al.*, 2011a; Mikolov *et al.*, 2013a).

15.2 Transfer Learning and Domain Adaptation

Transfer learning and domain adaptation refer to the situation where what has been learned in one setting (i.e., distribution P_1) is exploited to improve generalization in another setting (say distribution P_2). This generalizes the idea presented in the previous section, where we transferred representations between an unsupervised learning task and a supervised learning task.

In **transfer learning**, the learner must perform two or more different tasks, but we assume that many of the factors that explain the variations in P_1 are relevant to the variations that need to be captured for learning P_2. This is typically understood in a supervised learning context, where the input is the same but the target may be of a different nature. For example, we may learn about one set of visual categories, such as cats and dogs, in the first setting, then learn about a different set of visual categories, such as ants and wasps, in the second setting. If there is significantly more data in the first setting (sampled from P_1), then that may help to learn representations that are useful to quickly generalize from only very few examples drawn from P_2. Many visual categories *share* low-level notions of edges and visual shapes, the effects of geometric changes, changes in lighting, etc. In general, transfer learning, multi-task learning (section 7.7), and domain adaptation can be achieved via representation learning when there exist features that are useful for the different settings or tasks, corresponding to underlying factors that appear in more than one setting. This is illustrated in figure 7.2, with shared lower layers and task-dependent upper layers.

However, sometimes, what is shared among the different tasks is not the semantics of the input but the semantics of the output. For example, a speech recognition system needs to produce valid sentences at the output layer, but the earlier layers near the input may need to recognize very different versions of the same phonemes or sub-phonemic vocalizations depending on which person is speaking. In cases like these, it makes more sense to share the upper layers (near the output) of the neural network, and have a task-specific preprocessing, as

illustrated in figure 15.2.

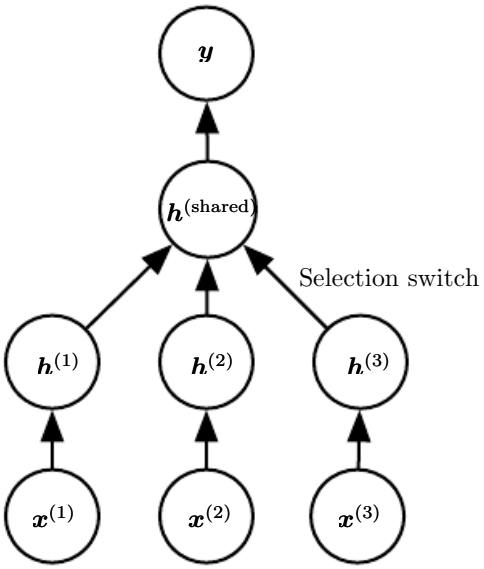


Figure 15.2: Example architecture for multi-task or transfer learning when the output variable $\mathbf{y}$ has the same semantics for all tasks while the input variable $\mathbf{x}$ has a different meaning (and possibly even a different dimension) for each task (or, for example, each user), called $\mathbf{x}^{(1)}$, $\mathbf{x}^{(2)}$ and $\mathbf{x}^{(3)}$ for three tasks. The lower levels (up to the selection switch) are task-specific, while the upper levels are shared. The lower levels learn to translate their task-specific input into a generic set of features.

In the related case of **domain adaptation**, the task (and the optimal input-to-output mapping) remains the same between each setting, but the input distribution is slightly different. For example, consider the task of sentiment analysis, which consists of determining whether a comment expresses positive or negative sentiment. Comments posted on the web come from many categories. A domain adaptation scenario can arise when a sentiment predictor trained on customer reviews of media content such as books, videos and music is later used to analyze comments about consumer electronics such as televisions or smartphones. One can imagine that there is an underlying function that tells whether any statement is positive, neutral or negative, but of course the vocabulary and style may vary from one domain to another, making it more difficult to generalize across domains. Simple unsupervised pretraining (with denoising autoencoders) has been found to be very successful for sentiment analysis with domain adaptation (Glorot *et al.*, 2011b).

A related problem is that of **concept drift**, which we can view as a form of transfer learning due to gradual changes in the data distribution over time. Both concept drift and transfer learning can be viewed as particular forms of

multi-task learning. While the phrase "multi-task learning" typically refers to supervised learning tasks, the more general notion of transfer learning is applicable to unsupervised learning and reinforcement learning as well.

In all of these cases, the objective is to take advantage of data from the first setting to extract information that may be useful when learning or even when directly making predictions in the second setting. The core idea of representation learning is that the same representation may be useful in both settings. Using the same representation in both settings allows the representation to benefit from the training data that is available for both tasks.

As mentioned before, unsupervised deep learning for transfer learning has found success in some machine learning competitions (Mesnil *et al.*, 2011; Goodfellow *et al.*, 2011). In the first of these competitions, the experimental setup is the following. Each participant is first given a dataset from the first setting (from distribution P_1), illustrating examples of some set of categories. The participants must use this to learn a good feature space (mapping the raw input to some representation), such that when we apply this learned transformation to inputs from the transfer setting (distribution P_2), a linear classifier can be trained and generalize well from very few labeled examples. One of the most striking results found in this competition is that as an architecture makes use of deeper and deeper representations (learned in a purely unsupervised way from data collected in the first setting, P_1), the learning curve on the new categories of the second (transfer) setting P_2 becomes much better. For deep representations, fewer labeled examples of the transfer tasks are necessary to achieve the apparently asymptotic generalization performance.

Two extreme forms of transfer learning are **one-shot learning** and **zero-shot learning**, sometimes also called **zero-data learning**. Only one labeled example of the transfer task is given for one-shot learning, while no labeled examples are given at all for the zero-shot learning task.

One-shot learning (Fei-Fei *et al.*, 2006) is possible because the representation learns to cleanly separate the underlying classes during the first stage. During the transfer learning stage, only one labeled example is needed to infer the label of many possible test examples that all cluster around the same point in representation space. This works to the extent that the factors of variation corresponding to these invariances have been cleanly separated from the other factors, in the learned representation space, and we have somehow learned which factors do and do not matter when discriminating objects of certain categories.

As an example of a zero-shot learning setting, consider the problem of having a learner read a large collection of text and then solve object recognition problems.

It may be possible to recognize a specific object class even without having seen an image of that object, if the text describes the object well enough. For example, having read that a cat has four legs and pointy ears, the learner might be able to guess that an image is a cat, without having seen a cat before.

Zero-data learning (Larochelle *et al.*, 2008) and zero-shot learning (Palatucci *et al.*, 2009; Socher *et al.*, 2013b) are only possible because additional information has been exploited during training. We can think of the zero-data learning scenario as including three random variables: the traditional inputs $\boldsymbol{x}$, the traditional outputs or targets $\boldsymbol{y}$, and an additional random variable describing the task, T. The model is trained to estimate the conditional distribution $p(\boldsymbol{y} \mid \boldsymbol{x}, T)$ where T is a description of the task we wish the model to perform. In our example of recognizing cats after having read about cats, the output is a binary variable y with $y = 1$ indicating "yes" and $y = 0$ indicating "no." The task variable T then represents questions to be answered such as "Is there a cat in this image?" If we have a training set containing unsupervised examples of objects that live in the same space as T, we may be able to infer the meaning of unseen instances of T. In our example of recognizing cats without having seen an image of the cat, it is important that we have had unlabeled text data containing sentences such as "cats have four legs" or "cats have pointy ears."

Zero-shot learning requires T to be represented in a way that allows some sort of generalization. For example, T cannot be just a one-hot code indicating an object category. Socher *et al.* (2013b) provide instead a distributed representation of object categories by using a learned word embedding for the word associated with each category.

A similar phenomenon happens in machine translation (Klementiev *et al.*, 2012; Mikolov *et al.*, 2013b; Gouws *et al.*, 2014): we have words in one language, and the relationships between words can be learned from unilingual corpora; on the other hand, we have translated sentences which relate words in one language with words in the other. Even though we may not have labeled examples translating word A in language X to word B in language Y, we can generalize and guess a translation for word A because we have learned a distributed representation for words in language X, a distributed representation for words in language Y, and created a link (possibly two-way) relating the two spaces, via training examples consisting of matched pairs of sentences in both languages. This transfer will be most successful if all three ingredients (the two representations and the relations between them) are learned jointly.

Zero-shot learning is a particular form of transfer learning. The same principle explains how one can perform **multi-modal learning**, capturing a representation

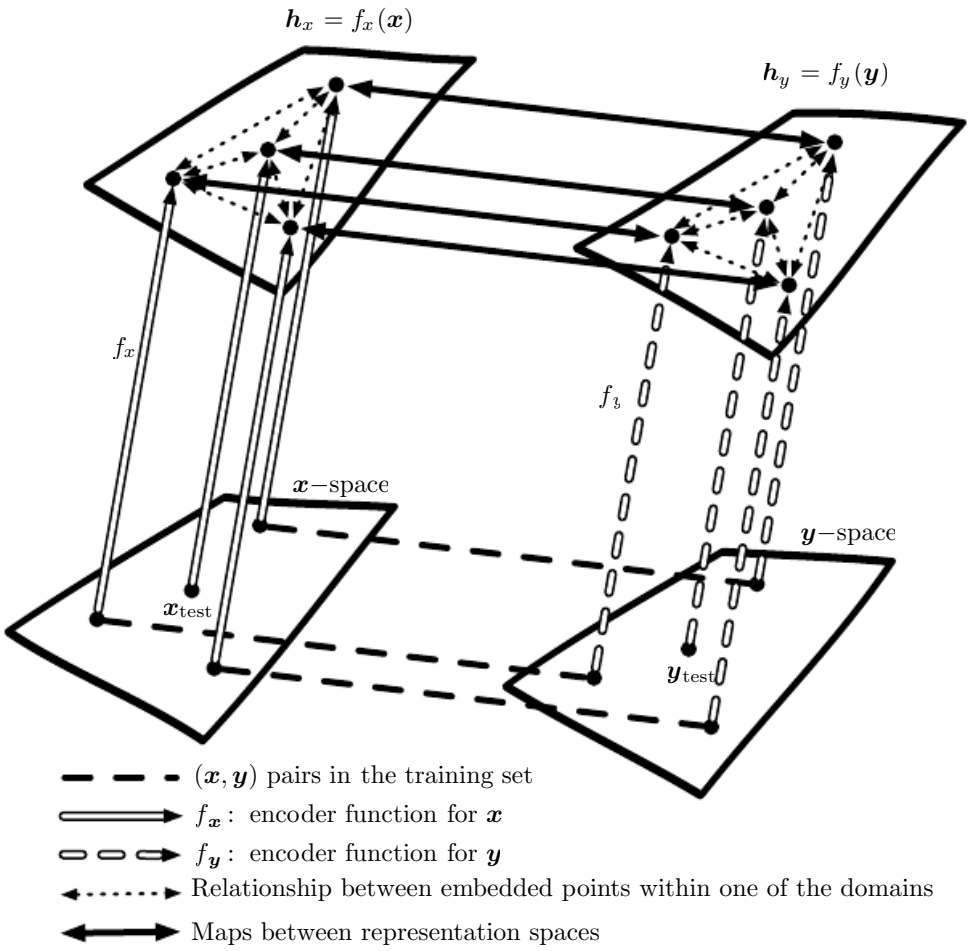


Figure 15.3: Transfer learning between two domains $\boldsymbol{x}$ and $\boldsymbol{y}$ enables zero-shot learning. Labeled or unlabeled examples of $\boldsymbol{x}$ allow one to learn a representation function f_x and similarly with examples of $\boldsymbol{y}$ to learn f_y. Each application of the f_x and f_y functions appears as an upward arrow, with the style of the arrows indicating which function is applied. Distance in $\boldsymbol{h}_x$ space provides a similarity metric between any pair of points in $\boldsymbol{x}$ space that may be more meaningful than distance in $\boldsymbol{x}$ space. Likewise, distance in $\boldsymbol{h}_y$ space provides a similarity metric between any pair of points in $\boldsymbol{y}$ space. Both of these similarity functions are indicated with dotted bidirectional arrows. Labeled examples (dashed horizontal lines) are pairs $(\boldsymbol{x}, \boldsymbol{y})$ which allow one to learn a one-way or two-way map (solid bidirectional arrow) between the representations $f_x(\boldsymbol{x})$ and the representations $f_y(\boldsymbol{y})$ and anchor these representations to each other. Zero-data learning is then enabled as follows. One can associate an image $\boldsymbol{x}_{\text{test}}$ to a word $\boldsymbol{y}_{\text{test}}$, even if no image of that word was ever presented, simply because word-representations $f_y(\boldsymbol{y}_{\text{test}})$ and image-representations $f_x(\boldsymbol{x}_{\text{test}})$ can be related to each other via the maps between representation spaces. It works because, although that image and that word were never paired, their respective feature vectors $f_x(\boldsymbol{x}_{\text{test}})$ and $f_y(\boldsymbol{y}_{\text{test}})$ have been related to each other. Figure inspired from suggestion by Hrant Khachatrian.

in one modality, a representation in the other, and the relationship (in general a joint distribution) between pairs $(\boldsymbol{x}, \boldsymbol{y})$ consisting of one observation $\boldsymbol{x}$ in one modality and another observation $\boldsymbol{y}$ in the other modality (Srivastava and Salakhutdinov, 2012). By learning all three sets of parameters (from $\boldsymbol{x}$ to its representation, from $\boldsymbol{y}$ to its representation, and the relationship between the two representations), concepts in one representation are anchored in the other, and vice-versa, allowing one to meaningfully generalize to new pairs. The procedure is illustrated in figure 15.3.

15.3 Semi-Supervised Disentangling of Causal Factors

An important question about representation learning is "what makes one representation better than another?" One hypothesis is that an ideal representation is one in which the features within the representation correspond to the underlying causes of the observed data, with separate features or directions in feature space corresponding to different causes, so that the representation disentangles the causes from one another. This hypothesis motivates approaches in which we first seek a good representation for $p(\boldsymbol{x})$. Such a representation may also be a good representation for computing $p(\boldsymbol{y} \mid \boldsymbol{x})$ if $\boldsymbol{y}$ is among the most salient causes of $\boldsymbol{x}$. This idea has guided a large amount of deep learning research since at least the 1990s (Becker and Hinton, 1992; Hinton and Sejnowski, 1999), in more detail. For other arguments about when semi-supervised learning can outperform pure supervised learning, we refer the reader to section 1.2 of Chapelle *et al.* (2006).

In other approaches to representation learning, we have often been concerned with a representation that is easy to model—for example, one whose entries are sparse, or independent from each other. A representation that cleanly separates the underlying causal factors may not necessarily be one that is easy to model. However, a further part of the hypothesis motivating semi-supervised learning via unsupervised representation learning is that for many AI tasks, these two properties coincide: once we are able to obtain the underlying explanations for what we observe, it generally becomes easy to isolate individual attributes from the others. Specifically, if a representation $\boldsymbol{h}$ represents many of the underlying causes of the observed $\boldsymbol{x}$, and the outputs $\boldsymbol{y}$ are among the most salient causes, then it is easy to predict $\boldsymbol{y}$ from $\boldsymbol{h}$.

First, let us see how semi-supervised learning can fail because unsupervised learning of $p(\mathbf{x})$ is of no help to learn $p(\mathbf{y} \mid \mathbf{x})$. Consider for example the case where $p(\mathbf{x})$ is uniformly distributed and we want to learn $f(\boldsymbol{x}) = \mathbb{E}[\mathbf{y} \mid \boldsymbol{x}]$. Clearly, observing a training set of $\boldsymbol{x}$ values alone gives us no information about $p(\mathbf{y} \mid \mathbf{x})$.

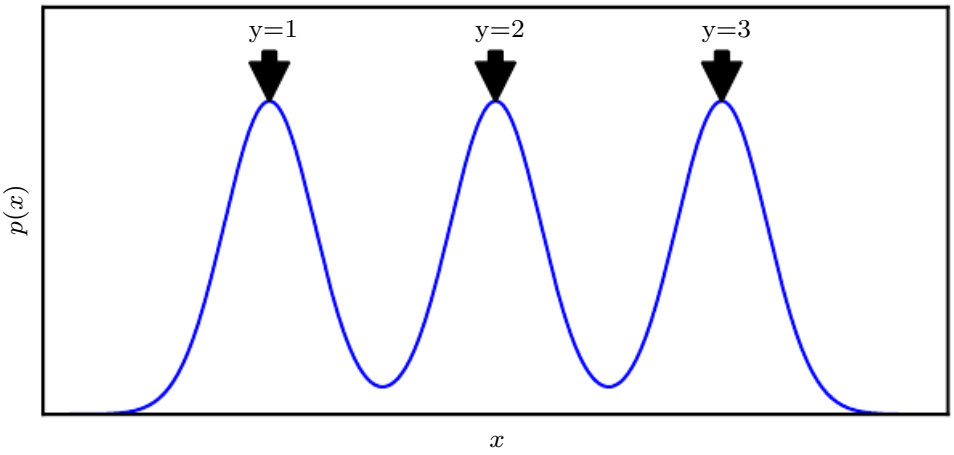


Figure 15.4: Example of a density over x that is a mixture over three components. The component identity is an underlying explanatory factor, y. Because the mixture components (e.g., natural object classes in image data) are statistically salient, just modeling $p(x)$ in an unsupervised way with no labeled example already reveals the factor y.

Next, let us see a simple example of how semi-supervised learning can succeed. Consider the situation where $\mathbf{x}$ arises from a mixture, with one mixture component per value of $\mathbf{y}$, as illustrated in figure 15.4. If the mixture components are well-separated, then modeling $p(\mathbf{x})$ reveals precisely where each component is, and a single labeled example of each class will then be enough to perfectly learn $p(\mathbf{y} \mid \mathbf{x})$. But more generally, what could make $p(\mathbf{y} \mid \mathbf{x})$ and $p(\mathbf{x})$ be tied together?

If $\mathbf{y}$ is closely associated with one of the causal factors of $\mathbf{x}$, then $p(\mathbf{x})$ and $p(\mathbf{y} \mid \mathbf{x})$ will be strongly tied, and unsupervised representation learning that tries to disentangle the underlying factors of variation is likely to be useful as a semi-supervised learning strategy.

Consider the assumption that $\mathbf{y}$ is one of the causal factors of $\mathbf{x}$, and let $\mathbf{h}$ represent all those factors. The true generative process can be conceived as structured according to this directed graphical model, with $\mathbf{h}$ as the parent of $\mathbf{x}$:

$$p(\mathbf{h}, \mathbf{x}) = p(\mathbf{x} \mid \mathbf{h})p(\mathbf{h}). \tag{15.1}$$

As a consequence, the data has marginal probability

$$p(\boldsymbol{x}) = \mathbb{E}_{\mathbf{h}} p(\boldsymbol{x} \mid \boldsymbol{h}). \tag{15.2}$$

From this straightforward observation, we conclude that the best possible model of $\mathbf{x}$ (from a generalization point of view) is the one that uncovers the above "true"

structure, with $\boldsymbol{h}$ as a latent variable that explains the observed variations in $\boldsymbol{x}$. The "ideal" representation learning discussed above should thus recover these latent factors. If $\mathbf{y}$ is one of these (or closely related to one of them), then it will be very easy to learn to predict $\mathbf{y}$ from such a representation. We also see that the conditional distribution of $\mathbf{y}$ given $\mathbf{x}$ is tied by Bayes' rule to the components in the above equation:

$$p(\mathbf{y} \mid \mathbf{x}) = \frac{p(\mathbf{x} \mid \mathbf{y})p(\mathbf{y})}{p(\mathbf{x})}. \tag{15.3}$$

Thus the marginal $p(\mathbf{x})$ is intimately tied to the conditional $p(\mathbf{y} \mid \mathbf{x})$ and knowledge of the structure of the former should be helpful to learn the latter. Therefore, in situations respecting these assumptions, semi-supervised learning should improve performance.

An important research problem regards the fact that most observations are formed by an extremely large number of underlying causes. Suppose $\mathbf{y} = \mathrm{h}_i$, but the unsupervised learner does not know which h_i. The brute force solution is for an unsupervised learner to learn a representation that captures *all* the reasonably salient generative factors h_j and disentangles them from each other, thus making it easy to predict $\mathbf{y}$ from $\mathbf{h}$, regardless of which h_i is associated with $\mathbf{y}$.

In practice, the brute force solution is not feasible because it is not possible to capture all or most of the factors of variation that influence an observation. For example, in a visual scene, should the representation always encode all of the smallest objects in the background? It is a well-documented psychological phenomenon that human beings fail to perceive changes in their environment that are not immediately relevant to the task they are performing—see, e.g., Simons and Levin (1998). An important research frontier in semi-supervised learning is determining *what* to encode in each situation. Currently, two of the main strategies for dealing with a large number of underlying causes are to use a supervised learning signal at the same time as the unsupervised learning signal so that the model will choose to capture the most relevant factors of variation, or to use much larger representations if using purely unsupervised learning.

An emerging strategy for unsupervised learning is to modify the definition of which underlying causes are most salient. Historically, autoencoders and generative models have been trained to optimize a fixed criterion, often similar to mean squared error. These fixed criteria determine which causes are considered salient. For example, mean squared error applied to the pixels of an image implicitly specifies that an underlying cause is only salient if it significantly changes the brightness of a large number of pixels. This can be problematic if the task we wish to solve involves interacting with small objects. See figure 15.5 for an example

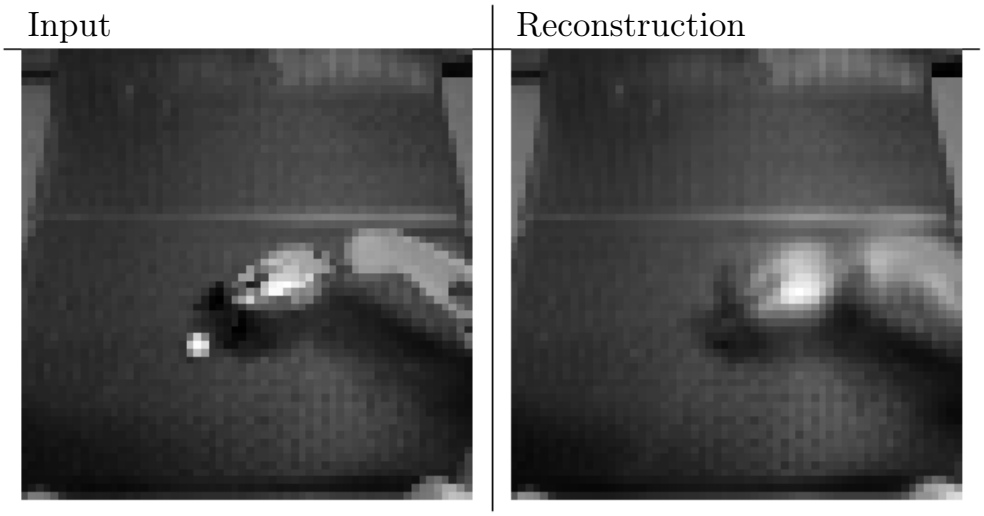


Figure 15.5: An autoencoder trained with mean squared error for a robotics task has failed to reconstruct a ping pong ball. The existence of the ping pong ball and all of its spatial coordinates are important underlying causal factors that generate the image and are relevant to the robotics task. Unfortunately, the autoencoder has limited capacity, and the training with mean squared error did not identify the ping pong ball as being salient enough to encode. Images graciously provided by Chelsea Finn.

of a robotics task in which an autoencoder has failed to learn to encode a small ping pong ball. This same robot is capable of successfully interacting with larger objects, such as baseballs, which are more salient according to mean squared error.

Other definitions of salience are possible. For example, if a group of pixels follow a highly recognizable pattern, even if that pattern does not involve extreme brightness or darkness, then that pattern could be considered extremely salient. One way to implement such a definition of salience is to use a recently developed approach called **generative adversarial networks** (Goodfellow *et al.*, 2014c). In this approach, a generative model is trained to fool a feedforward classifier. The feedforward classifier attempts to recognize all samples from the generative model as being fake, and all samples from the training set as being real. In this framework, any structured pattern that the feedforward network can recognize is highly salient. The generative adversarial network will be described in more detail in section 20.10.4. For the purposes of the present discussion, it is sufficient to understand that they *learn* how to determine what is salient. Lotter *et al.* (2015) showed that models trained to generate images of human heads will often neglect to generate the ears when trained with mean squared error, but will successfully generate the ears when trained with the adversarial framework. Because the ears are not extremely bright or dark compared to the surrounding skin, they are not especially salient according to mean squared error loss, but their highly

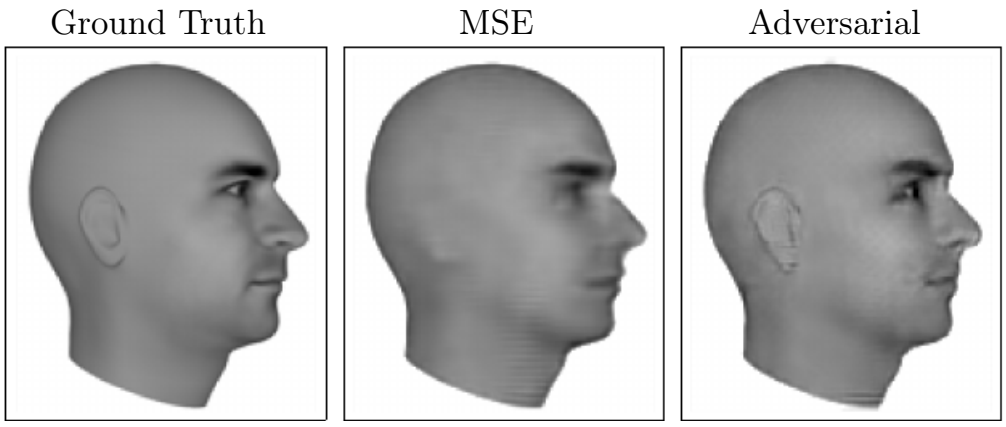


Figure 15.6: Predictive generative networks provide an example of the importance of learning which features are salient. In this example, the predictive generative network has been trained to predict the appearance of a 3-D model of a human head at a specific viewing angle. *(Left)*Ground truth. This is the correct image, that the network should emit. *(Center)*Image produced by a predictive generative network trained with mean squared error alone. Because the ears do not cause an extreme difference in brightness compared to the neighboring skin, they were not sufficiently salient for the model to learn to represent them. *(Right)*Image produced by a model trained with a combination of mean squared error and adversarial loss. Using this learned cost function, the ears are salient because they follow a predictable pattern. Learning which underlying causes are important and relevant enough to model is an important active area of research. Figures graciously provided by Lotter *et al.* (2015).

recognizable shape and consistent position means that a feedforward network can easily learn to detect them, making them highly salient under the generative adversarial framework. See figure 15.6 for example images. Generative adversarial networks are only one step toward determining which factors should be represented. We expect that future research will discover better ways of determining which factors to represent, and develop mechanisms for representing different factors depending on the task.

A benefit of learning the underlying causal factors, as pointed out by Schölkopf *et al.* (2012), is that if the true generative process has $\mathbf{x}$ as an effect and $\mathbf{y}$ as a cause, then modeling $p(\mathbf{x} \mid \mathbf{y})$ is robust to changes in $p(\mathbf{y})$. If the cause-effect relationship was reversed, this would not be true, since by Bayes' rule, $p(\mathbf{x} \mid \mathbf{y})$ would be sensitive to changes in $p(\mathbf{y})$. Very often, when we consider changes in distribution due to different domains, temporal non-stationarity, or changes in the nature of the task, *the causal mechanisms remain invariant* (the laws of the universe are constant) while the marginal distribution over the underlying causes can change. Hence, better generalization and robustness to all kinds of changes can

be expected via learning a generative model that attempts to recover the causal factors $\mathbf{h}$ and $p(\mathbf{x} \mid \mathbf{h})$.

15.4 Distributed Representation

Distributed representations of concepts—representations composed of many elements that can be set separately from each other—are one of the most important tools for representation learning. Distributed representations are powerful because they can use n features with k values to describe k^n different concepts. As we have seen throughout this book, both neural networks with multiple hidden units and probabilistic models with multiple latent variables make use of the strategy of distributed representation. We now introduce an additional observation. Many deep learning algorithms are motivated by the assumption that the hidden units can learn to represent the underlying causal factors that explain the data, as discussed in section 15.3. Distributed representations are natural for this approach, because each direction in representation space can correspond to the value of a different underlying configuration variable.

An example of a distributed representation is a vector of n binary features, which can take 2^n configurations, each potentially corresponding to a different region in input space, as illustrated in figure 15.7. This can be compared with a *symbolic representation*, where the input is associated with a single symbol or category. If there are n symbols in the dictionary, one can imagine n feature detectors, each corresponding to the detection of the presence of the associated category. In that case only n different configurations of the representation space are possible, carving n different regions in input space, as illustrated in figure 15.8. Such a symbolic representation is also called a one-hot representation, since it can be captured by a binary vector with n bits that are mutually exclusive (only one of them can be active). A symbolic representation is a specific example of the broader class of non-distributed representations, which are representations that may contain many entries but without significant meaningful separate control over each entry.

Examples of learning algorithms based on non-distributed representations include:

- Clustering methods, including the k-means algorithm: each input point is assigned to exactly one cluster.
- k-nearest neighbors algorithms: one or a few templates or prototype examples are associated with a given input. In the case of $k > 1$, there are multiple

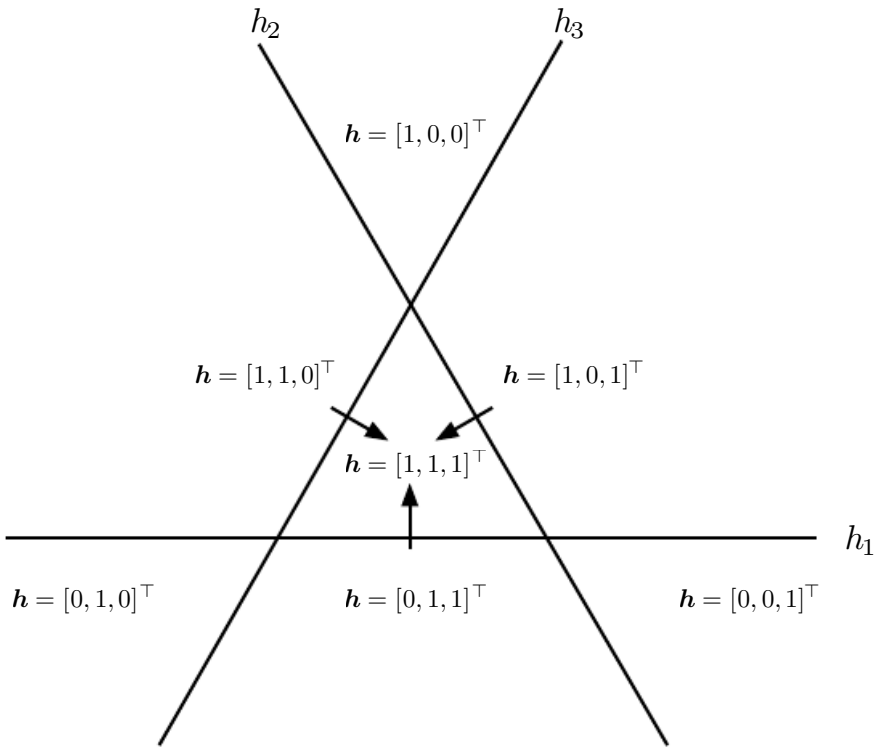


Figure 15.7: Illustration of how a learning algorithm based on a distributed representation breaks up the input space into regions. In this example, there are three binary features h_1, h_2, and h_3. Each feature is defined by thresholding the output of a learned, linear transformation. Each feature divides $\mathbb{R}^2$ into two half-planes. Let h_i^+ be the set of input points for which $h_i = 1$ and h_i^- be the set of input points for which $h_i = 0$. In this illustration, each line represents the decision boundary for one h_i, with the corresponding arrow pointing to the h_i^+ side of the boundary. The representation as a whole takes on a unique value at each possible intersection of these half-planes. For example, the representation value $[1, 1, 1]^\top$ corresponds to the region $h_1^+ \cap h_2^+ \cap h_3^+$. Compare this to the non-distributed representations in figure 15.8. In the general case of d input dimensions, a distributed representation divides $\mathbb{R}^d$ by intersecting half-spaces rather than half-planes. The distributed representation with n features assigns unique codes to $O(n^d)$ different regions, while the nearest neighbor algorithm with n examples assigns unique codes to only n regions. The distributed representation is thus able to distinguish exponentially many more regions than the non-distributed one. Keep in mind that not all $\boldsymbol{h}$ values are feasible (there is no $\boldsymbol{h} = \mathbf{0}$ in this example) and that a linear classifier on top of the distributed representation is not able to assign different class identities to every neighboring region; even a deep linear-threshold network has a VC dimension of only $O(w \log w)$ where w is the number of weights (Sontag, 1998). The combination of a powerful representation layer and a weak classifier layer can be a strong regularizer; a classifier trying to learn the concept of "person" versus "not a person" does not need to assign a different class to an input represented as "woman with glasses" than it assigns to an input represented as "man without glasses." This capacity constraint encourages each classifier to focus on few h_i and encourages $\boldsymbol{h}$ to learn to represent the classes in a linearly separable way.

values describing each input, but they can not be controlled separately from each other, so this does not qualify as a true distributed representation.

- Decision trees: only one leaf (and the nodes on the path from root to leaf) is activated when an input is given.
- Gaussian mixtures and mixtures of experts: the templates (cluster centers) or experts are now associated with a *degree* of activation. As with the k-nearest neighbors algorithm, each input is represented with multiple values, but those values cannot readily be controlled separately from each other.
- Kernel machines with a Gaussian kernel (or other similarly local kernel): although the degree of activation of each "support vector" or template example is now continuous-valued, the same issue arises as with Gaussian mixtures.
- Language or translation models based on n-grams. The set of contexts (sequences of symbols) is partitioned according to a tree structure of suffixes. A leaf may correspond to the last two words being w_1 and w_2, for example. Separate parameters are estimated for each leaf of the tree (with some sharing being possible).

For some of these non-distributed algorithms, the output is not constant by parts but instead interpolates between neighboring regions. The relationship between the number of parameters (or examples) and the number of regions they can define remains linear.

An important related concept that distinguishes a distributed representation from a symbolic one is that *generalization arises due to shared attributes* between different concepts. As pure symbols, "`cat`" and "`dog`" are as far from each other as any other two symbols. However, if one associates them with a meaningful distributed representation, then many of the things that can be said about cats can generalize to dogs and vice-versa. For example, our distributed representation may contain entries such as "`has_fur`" or "`number_of_legs`" that have the same value for the embedding of both "`cat`" and "`dog`." Neural language models that operate on distributed representations of words generalize much better than other models that operate directly on one-hot representations of words, as discussed in section 12.4. Distributed representations induce a rich *similarity space*, in which semantically close concepts (or inputs) are close in distance, a property that is absent from purely symbolic representations.

When and why can there be a statistical advantage from using a distributed representation as part of a learning algorithm? Distributed representations can

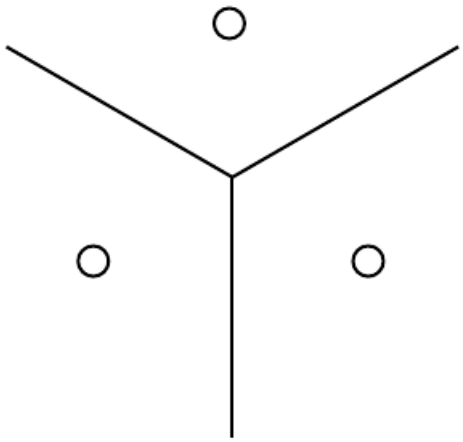


Figure 15.8: Illustration of how the nearest neighbor algorithm breaks up the input space into different regions. The nearest neighbor algorithm provides an example of a learning algorithm based on a non-distributed representation. Different non-distributed algorithms may have different geometry, but they typically break the input space into regions, *with a separate set of parameters for each region*. The advantage of a non-distributed approach is that, given enough parameters, it can fit the training set without solving a difficult optimization algorithm, because it is straightforward to choose a different output *independently* for each region. The disadvantage is that such non-distributed models generalize only locally via the smoothness prior, making it difficult to learn a complicated function with more peaks and troughs than the available number of examples. Contrast this with a distributed representation, figure 15.7.

have a statistical advantage when an apparently complicated structure can be compactly represented using a small number of parameters. Some traditional non-distributed learning algorithms generalize only due to the smoothness assumption, which states that if $u \approx v$, then the target function f to be learned has the property that $f(u) \approx f(v)$, in general. There are many ways of formalizing such an assumption, but the end result is that if we have an example (x, y) for which we know that $f(x) \approx y$, then we choose an estimator $\hat{f}$ that approximately satisfies these constraints while changing as little as possible when we move to a nearby input $x + \epsilon$. This assumption is clearly very useful, but it suffers from the curse of dimensionality: in order to learn a target function that increases and decreases many times in many different regions,[1] we may need a number of examples that is at least as large as the number of distinguishable regions. One can think of each of these regions as a category or symbol: by having a separate degree of freedom for each symbol (or region), we can learn an arbitrary decoder mapping from symbol to value. However, this does not allow us to generalize to new symbols for new regions.

If we are lucky, there may be some regularity in the target function, besides being smooth. For example, a convolutional network with max-pooling can recognize an object regardless of its location in the image, even though spatial translation of the object may not correspond to smooth transformations in the input space.

Let us examine a special case of a distributed representation learning algorithm, that extracts binary features by thresholding linear functions of the input. Each binary feature in this representation divides $\mathbb{R}^d$ into a pair of half-spaces, as illustrated in figure 15.7. The exponentially large number of intersections of n of the corresponding half-spaces determines how many regions this distributed representation learner can distinguish. How many regions are generated by an arrangement of n hyperplanes in $\mathbb{R}^d$? By applying a general result concerning the intersection of hyperplanes (Zaslavsky, 1975), one can show (Pascanu *et al.*, 2014b) that the number of regions this binary feature representation can distinguish is

$$\sum_{j=0}^{d} \binom{n}{j} = O(n^d). \tag{15.4}$$

Therefore, we see a growth that is exponential in the input size and polynomial in the number of hidden units.

[1] Potentially, we may want to learn a function whose behavior is distinct in exponentially many regions: in a d-dimensional space with at least 2 different values to distinguish per dimension, we might want f to differ in 2^d different regions, requiring $O(2^d)$ training examples.

This provides a geometric argument to explain the generalization power of distributed representation: with $O(nd)$ parameters (for n linear-threshold features in $\mathbb{R}^d$) we can distinctly represent $O(n^d)$ regions in input space. If instead we made no assumption at all about the data, and used a representation with one unique symbol for each region, and separate parameters for each symbol to recognize its corresponding portion of $\mathbb{R}^d$, then specifying $O(n^d)$ regions would require $O(n^d)$ examples. More generally, the argument in favor of the distributed representation could be extended to the case where instead of using linear threshold units we use nonlinear, possibly continuous, feature extractors for each of the attributes in the distributed representation. The argument in this case is that if a parametric transformation with k parameters can learn about r regions in input space, with $k \ll r$, and if obtaining such a representation was useful to the task of interest, then we could potentially generalize much better in this way than in a non-distributed setting where we would need $O(r)$ examples to obtain the same features and associated partitioning of the input space into r regions. Using fewer parameters to represent the model means that we have fewer parameters to fit, and thus require far fewer training examples to generalize well.

A further part of the argument for why models based on distributed representations generalize well is that their capacity remains limited despite being able to distinctly encode so many different regions. For example, the VC dimension of a neural network of linear threshold units is only $O(w \log w)$, where w is the number of weights (Sontag, 1998). This limitation arises because, while we can assign very many unique codes to representation space, we cannot use absolutely all of the code space, nor can we learn arbitrary functions mapping from the representation space $\boldsymbol{h}$ to the output $\boldsymbol{y}$ using a linear classifier. The use of a distributed representation combined with a linear classifier thus expresses a prior belief that the classes to be recognized are linearly separable as a function of the underlying causal factors captured by $\boldsymbol{h}$. We will typically want to learn categories such as the set of all images of all green objects or the set of all images of cars, but not categories that require nonlinear, XOR logic. For example, we typically do not want to partition the data into the set of all red cars and green trucks as one class and the set of all green cars and red trucks as another class.

The ideas discussed so far have been abstract, but they may be experimentally validated. Zhou *et al.* (2015) find that hidden units in a deep convolutional network trained on the ImageNet and Places benchmark datasets learn features that are very often interpretable, corresponding to a label that humans would naturally assign. In practice it is certainly not always the case that hidden units learn something that has a simple linguistic name, but it is interesting to see this emerge near the top levels of the best computer vision deep networks. What such features have in

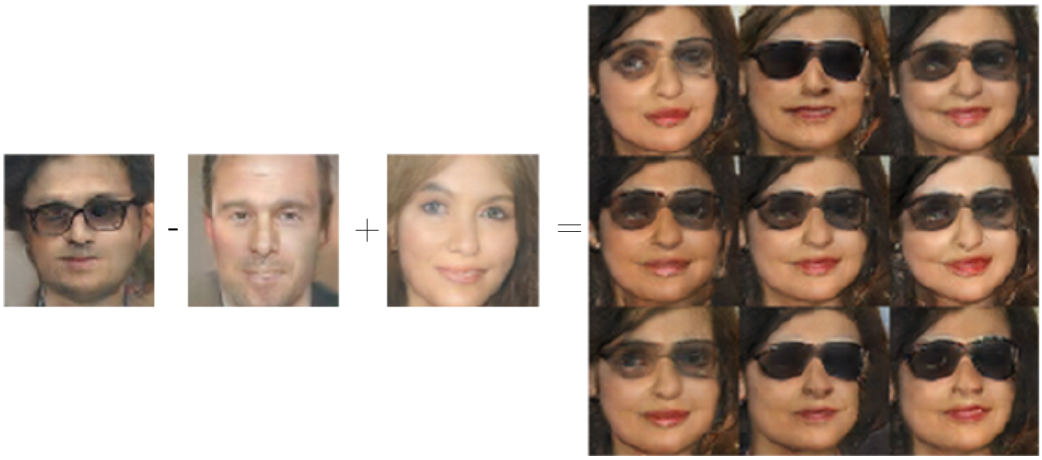


Figure 15.9: A generative model has learned a distributed representation that disentangles the concept of gender from the concept of wearing glasses. If we begin with the representation of the concept of a man with glasses, then subtract the vector representing the concept of a man without glasses, and finally add the vector representing the concept of a woman without glasses, we obtain the vector representing the concept of a woman with glasses. The generative model correctly decodes all of these representation vectors to images that may be recognized as belonging to the correct class. Images reproduced with permission from Radford *et al.* (2015).

common is that one could imagine *learning about each of them without having to see all the configurations of all the others*. Radford *et al.* (2015) demonstrated that a generative model can learn a representation of images of faces, with separate directions in representation space capturing different underlying factors of variation. Figure 15.9 demonstrates that one direction in representation space corresponds to whether the person is male or female, while another corresponds to whether the person is wearing glasses. These features were discovered automatically, not fixed a priori. There is no need to have labels for the hidden unit classifiers: gradient descent on an objective function of interest naturally learns semantically interesting features, so long as the task requires such features. We can learn about the distinction between male and female, or about the presence or absence of glasses, without having to characterize all of the configurations of the $n - 1$ other features by examples covering all of these combinations of values. This form of statistical separability is what allows one to generalize to new configurations of a person's features that have never been seen during training.

15.5 Exponential Gains from Depth

We have seen in section 6.4.1 that multilayer perceptrons are universal approximators, and that some functions can be represented by exponentially smaller deep networks compared to shallow networks. This decrease in model size leads to improved statistical efficiency. In this section, we describe how similar results apply more generally to other kinds of models with distributed hidden representations.

In section 15.4, we saw an example of a generative model that learned about the explanatory factors underlying images of faces, including the person's gender and whether they are wearing glasses. The generative model that accomplished this task was based on a deep neural network. It would not be reasonable to expect a shallow network, such as a linear network, to learn the complicated relationship between these abstract explanatory factors and the pixels in the image. In this and other AI tasks, the factors that can be chosen almost independently from each other yet still correspond to meaningful inputs are more likely to be very high-level and related in highly nonlinear ways to the input. We argue that this demands *deep* distributed representations, where the higher level features (seen as functions of the input) or factors (seen as generative causes) are obtained through the composition of many nonlinearities.

It has been proven in many different settings that organizing computation through the composition of many nonlinearities and a hierarchy of reused features can give an exponential boost to statistical efficiency, on top of the exponential boost given by using a distributed representation. Many kinds of networks (e.g., with saturating nonlinearities, Boolean gates, sum/products, or RBF units) with a single hidden layer can be shown to be universal approximators. A model family that is a universal approximator can approximate a large class of functions (including all continuous functions) up to any non-zero tolerance level, given enough hidden units. However, the required number of hidden units may be very large. Theoretical results concerning the expressive power of deep architectures state that there are families of functions that can be represented efficiently by an architecture of depth k, but would require an exponential number of hidden units (with respect to the input size) with insufficient depth (depth 2 or depth $k-1$).

In section 6.4.1, we saw that deterministic feedforward networks are universal approximators of functions. Many structured probabilistic models with a single hidden layer of latent variables, including restricted Boltzmann machines and deep belief networks, are universal approximators of probability distributions (Le Roux and Bengio, 2008, 2010; Montúfar and Ay, 2011; Montúfar, 2014; Krause *et al.*, 2013).

In section 6.4.1, we saw that a sufficiently deep feedforward network can have an exponential advantage over a network that is too shallow. Such results can also be obtained for other models such as probabilistic models. One such probabilistic model is the **sum-product network** or SPN (Poon and Domingos, 2011). These models use polynomial circuits to compute the probability distribution over a set of random variables. Delalleau and Bengio (2011) showed that there exist probability distributions for which a minimum depth of SPN is required to avoid needing an exponentially large model. Later, Martens and Medabalimi (2014) showed that there are significant differences between every two finite depths of SPN, and that some of the constraints used to make SPNs tractable may limit their representational power.

Another interesting development is a set of theoretical results for the expressive power of families of deep circuits related to convolutional nets, highlighting an exponential advantage for the deep circuit even when the shallow circuit is allowed to only approximate the function computed by the deep circuit (Cohen *et al.*, 2015). By comparison, previous theoretical work made claims regarding only the case where the shallow circuit must exactly replicate particular functions.

15.6 Providing Clues to Discover Underlying Causes

To close this chapter, we come back to one of our original questions: what makes one representation better than another? One answer, first introduced in section 15.3, is that an ideal representation is one that disentangles the underlying causal factors of variation that generated the data, especially those factors that are relevant to our applications. Most strategies for representation learning are based on introducing clues that help the learning to find these underlying factors of variations. The clues can help the learner separate these observed factors from the others. Supervised learning provides a very strong clue: a label $\boldsymbol{y}$, presented with each $\boldsymbol{x}$, that usually specifies the value of at least one of the factors of variation directly. More generally, to make use of abundant unlabeled data, representation learning makes use of other, less direct, hints about the underlying factors. These hints take the form of implicit prior beliefs that we, the designers of the learning algorithm, impose in order to guide the learner. Results such as the no free lunch theorem show that regularization strategies are necessary to obtain good generalization. While it is impossible to find a universally superior regularization strategy, one goal of deep learning is to find a set of fairly generic regularization strategies that are applicable to a wide variety of AI tasks, similar to the tasks that people and animals are able to solve.

We provide here a list of these generic regularization strategies. The list is clearly not exhaustive, but gives some concrete examples of ways that learning algorithms can be encouraged to discover features that correspond to underlying factors. This list was introduced in section 3.1 of Bengio *et al.* (2013d) and has been partially expanded here.

- *Smoothness*: This is the assumption that $f(\boldsymbol{x} + \epsilon \boldsymbol{d}) \approx f(\boldsymbol{x})$ for unit $\boldsymbol{d}$ and small ϵ. This assumption allows the learner to generalize from training examples to nearby points in input space. Many machine learning algorithms leverage this idea, but it is insufficient to overcome the curse of dimensionality.

- *Linearity*: Many learning algorithms assume that relationships between some variables are linear. This allows the algorithm to make predictions even very far from the observed data, but can sometimes lead to overly extreme predictions. Most simple machine learning algorithms that do not make the smoothness assumption instead make the linearity assumption. These are in fact different assumptions—linear functions with large weights applied to high-dimensional spaces may not be very smooth. See Goodfellow *et al.* (2014b) for a further discussion of the limitations of the linearity assumption.

- *Multiple explanatory factors*: Many representation learning algorithms are motivated by the assumption that the data is generated by multiple underlying explanatory factors, and that most tasks can be solved easily given the state of each of these factors. Section 15.3 describes how this view motivates semi-supervised learning via representation learning. Learning the structure of $p(\boldsymbol{x})$ requires learning some of the same features that are useful for modeling $p(\boldsymbol{y} \mid \boldsymbol{x})$ because both refer to the same underlying explanatory factors. Section 15.4 describes how this view motivates the use of distributed representations, with separate directions in representation space corresponding to separate factors of variation.

- *Causal factors*: the model is constructed in such a way that it treats the factors of variation described by the learned representation $\boldsymbol{h}$ as the causes of the observed data $\boldsymbol{x}$, and not vice-versa. As discussed in section 15.3, this is advantageous for semi-supervised learning and makes the learned model more robust when the distribution over the underlying causes changes or when we use the model for a new task.

- *Depth*, or *a hierarchical organization of explanatory factors*: High-level, abstract concepts can be defined in terms of simple concepts, forming a hierarchy. From another point of view, the use of a deep architecture

expresses our belief that the task should be accomplished via a multi-step program, with each step referring back to the output of the processing accomplished via previous steps.

- *Shared factors across tasks*: In the context where we have many tasks, corresponding to different y_i variables sharing the same input $\mathbf{x}$ or where each task is associated with a subset or a function $f^{(i)}(\mathbf{x})$ of a global input $\mathbf{x}$, the assumption is that each y_i is associated with a different subset from a common pool of relevant factors $\mathbf{h}$. Because these subsets overlap, learning all the $P(y_i \mid \mathbf{x})$ via a shared intermediate representation $P(\mathbf{h} \mid \mathbf{x})$ allows sharing of statistical strength between the tasks.

- *Manifolds*: Probability mass concentrates, and the regions in which it concentrates are locally connected and occupy a tiny volume. In the continuous case, these regions can be approximated by low-dimensional manifolds with a much smaller dimensionality than the original space where the data lives. Many machine learning algorithms behave sensibly only on this manifold (Goodfellow *et al.*, 2014b). Some machine learning algorithms, especially autoencoders, attempt to explicitly learn the structure of the manifold.

- *Natural clustering*: Many machine learning algorithms assume that each connected manifold in the input space may be assigned to a single class. The data may lie on many disconnected manifolds, but the class remains constant within each one of these. This assumption motivates a variety of learning algorithms, including tangent propagation, double backprop, the manifold tangent classifier and adversarial training.

- *Temporal and spatial coherence*: Slow feature analysis and related algorithms make the assumption that the most important explanatory factors change slowly over time, or at least that it is easier to predict the true underlying explanatory factors than to predict raw observations such as pixel values. See section 13.3 for further description of this approach.

- *Sparsity*: Most features should presumably not be relevant to describing most inputs—there is no need to use a feature that detects elephant trunks when representing an image of a cat. It is therefore reasonable to impose a prior that any feature that can be interpreted as "present" or "absent" should be absent most of the time.

- *Simplicity of Factor Dependencies*: In good high-level representations, the factors are related to each other through simple dependencies. The simplest

possible is marginal independence, $P(\mathbf{h}) = \prod_i P(\mathbf{h}_i)$, but linear dependencies or those captured by a shallow autoencoder are also reasonable assumptions. This can be seen in many laws of physics, and is assumed when plugging a linear predictor or a factorized prior on top of a learned representation.

The concept of representation learning ties together all of the many forms of deep learning. Feedforward and recurrent networks, autoencoders and deep probabilistic models all learn and exploit representations. Learning the best possible representation remains an exciting avenue of research.

Chapter 16

Structured Probabilistic Models for Deep Learning

Deep learning draws upon many modeling formalisms that researchers can use to guide their design efforts and describe their algorithms. One of these formalisms is the idea of **structured probabilistic models**. We have already discussed structured probabilistic models briefly in section 3.14. That brief presentation was sufficient to understand how to use structured probabilistic models as a language to describe some of the algorithms in part II. Now, in part III, structured probabilistic models are a key ingredient of many of the most important research topics in deep learning. In order to prepare to discuss these research ideas, this chapter describes structured probabilistic models in much greater detail. This chapter is intended to be self-contained; the reader does not need to review the earlier introduction before continuing with this chapter.

A structured probabilistic model is a way of describing a probability distribution, using a graph to describe which random variables in the probability distribution interact with each other directly. Here we use "graph" in the graph theory sense—a set of vertices connected to one another by a set of edges. Because the structure of the model is defined by a graph, these models are often also referred to as **graphical models**.

The graphical models research community is large and has developed many different models, training algorithms, and inference algorithms. In this chapter, we provide basic background on some of the most central ideas of graphical models, with an emphasis on the concepts that have proven most useful to the deep learning research community. If you already have a strong background in graphical models, you may wish to skip most of this chapter. However, even a graphical model expert

may benefit from reading the final section of this chapter, section 16.7, in which we highlight some of the unique ways that graphical models are used for deep learning algorithms. Deep learning practitioners tend to use very different model structures, learning algorithms and inference procedures than are commonly used by the rest of the graphical models research community. In this chapter, we identify these differences in preferences and explain the reasons for them.

In this chapter we first describe the challenges of building large-scale probabilistic models. Next, we describe how to use a graph to describe the structure of a probability distribution. While this approach allows us to overcome many challenges, it is not without its own complications. One of the major difficulties in graphical modeling is understanding which variables need to be able to interact directly, i.e., which graph structures are most suitable for a given problem. We outline two approaches to resolving this difficulty by learning about the dependencies in section 16.5. Finally, we close with a discussion of the unique emphasis that deep learning practitioners place on specific approaches to graphical modeling in section 16.7.

16.1 The Challenge of Unstructured Modeling

The goal of deep learning is to scale machine learning to the kinds of challenges needed to solve artificial intelligence. This means being able to understand high-dimensional data with rich structure. For example, we would like AI algorithms to be able to understand natural images,[1] audio waveforms representing speech, and documents containing multiple words and punctuation characters.

Classification algorithms can take an input from such a rich high-dimensional distribution and summarize it with a categorical label—what object is in a photo, what word is spoken in a recording, what topic a document is about. The process of classification discards most of the information in the input and produces a single output (or a probability distribution over values of that single output). The classifier is also often able to ignore many parts of the input. For example, when recognizing an object in a photo, it is usually possible to ignore the background of the photo.

It is possible to ask probabilistic models to do many other tasks. These tasks are often more expensive than classification. Some of them require producing multiple output values. Most require a complete understanding of the entire structure of

[1] A **natural image** is an image that might be captured by a camera in a reasonably ordinary environment, as opposed to a synthetically rendered image, a screenshot of a web page, etc.

the input, with no option to ignore sections of it. These tasks include the following:

- **Density estimation**: given an input $\boldsymbol{x}$, the machine learning system returns an estimate of the true density $p(\boldsymbol{x})$ under the data generating distribution. This requires only a single output, but it does require a complete understanding of the entire input. If even one element of the vector is unusual, the system must assign it a low probability.

- **Denoising**: given a damaged or incorrectly observed input $\tilde{\boldsymbol{x}}$, the machine learning system returns an estimate of the original or correct $\boldsymbol{x}$. For example, the machine learning system might be asked to remove dust or scratches from an old photograph. This requires multiple outputs (every element of the estimated clean example $\boldsymbol{x}$) and an understanding of the entire input (since even one damaged area will still reveal the final estimate as being damaged).

- **Missing value imputation**: given the observations of some elements of $\boldsymbol{x}$, the model is asked to return estimates of or a probability distribution over some or all of the unobserved elements of $\boldsymbol{x}$. This requires multiple outputs. Because the model could be asked to restore any of the elements of $\boldsymbol{x}$, it must understand the entire input.

- **Sampling**: the model generates new samples from the distribution $p(\boldsymbol{x})$. Applications include speech synthesis, i.e. producing new waveforms that sound like natural human speech. This requires multiple output values and a good model of the entire input. If the samples have even one element drawn from the wrong distribution, then the sampling process is wrong.

For an example of a sampling task using small natural images, see figure 16.1.

Modeling a rich distribution over thousands or millions of random variables is a challenging task, both computationally and statistically. Suppose we only wanted to model binary variables. This is the simplest possible case, and yet already it seems overwhelming. For a small, 32×32 pixel color (RGB) image, there are 2^{3072} possible binary images of this form. This number is over 10^{800} times larger than the estimated number of atoms in the universe.

In general, if we wish to model a distribution over a random vector $\mathbf{x}$ containing n discrete variables capable of taking on k values each, then the naive approach of representing $P(\mathbf{x})$ by storing a lookup table with one probability value per possible outcome requires k^n parameters!

This is not feasible for several reasons:

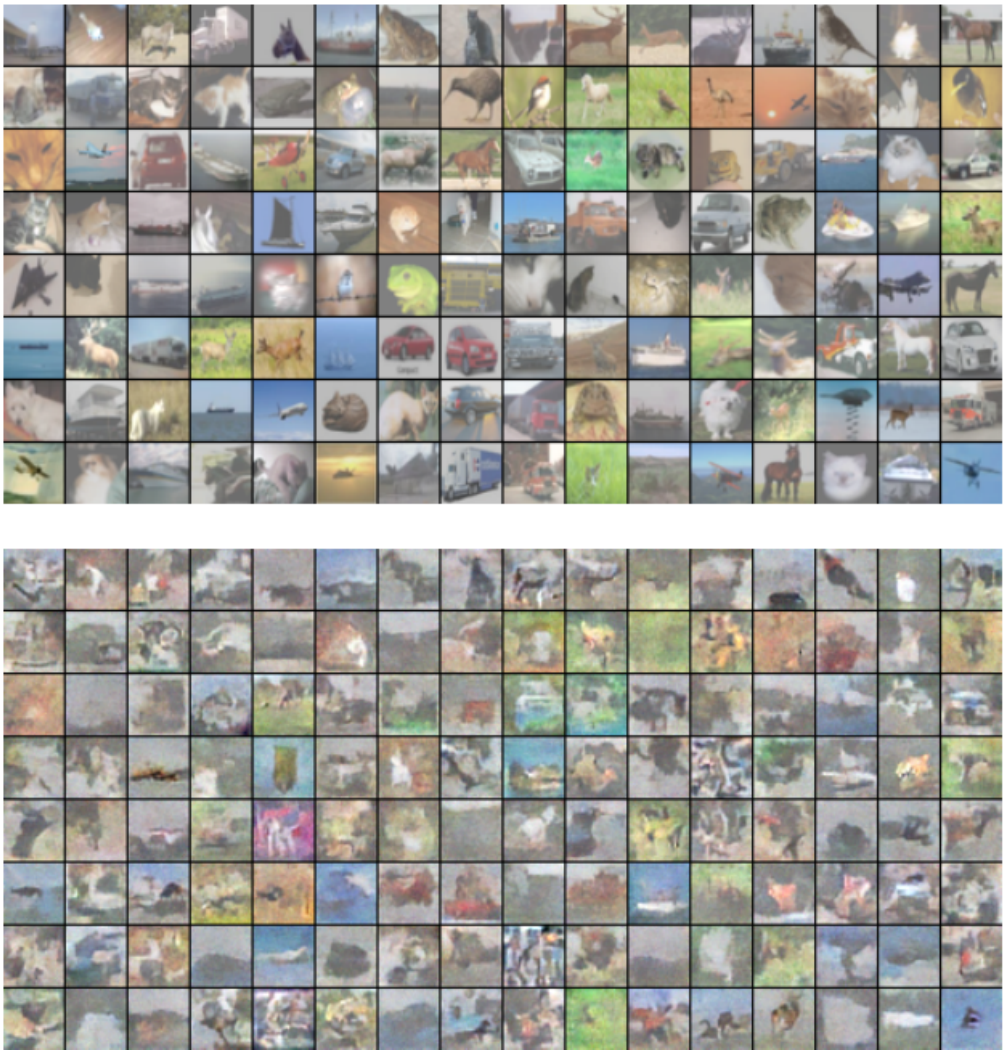


Figure 16.1: Probabilistic modeling of natural images. *(Top)*Example 32× 32 pixel color images from the CIFAR-10 dataset (Krizhevsky and Hinton, 2009). *(Bottom)*Samples drawn from a structured probabilistic model trained on this dataset. Each sample appears at the same position in the grid as the training example that is closest to it in Euclidean space. This comparison allows us to see that the model is truly synthesizing new images, rather than memorizing the training data. Contrast of both sets of images has been adjusted for display. Figure reproduced with permission from Courville *et al.* (2011).

- *Memory: the cost of storing the representation*: For all but very small values of n and k, representing the distribution as a table will require too many values to store.

- *Statistical efficiency*: As the number of parameters in a model increases, so does the amount of training data needed to choose the values of those parameters using a statistical estimator. Because the table-based model has an astronomical number of parameters, it will require an astronomically large training set to fit accurately. Any such model will overfit the training set very badly unless additional assumptions are made linking the different entries in the table (for example, like in back-off or smoothed n-gram models, section 12.4.1).

- *Runtime: the cost of inference*: Suppose we want to perform an inference task where we use our model of the joint distribution $P(\mathbf{x})$ to compute some other distribution, such as the marginal distribution $P(\mathrm{x}_1)$ or the conditional distribution $P(\mathrm{x}_2 \mid \mathrm{x}_1)$. Computing these distributions will require summing across the entire table, so the runtime of these operations is as high as the intractable memory cost of storing the model.

- *Runtime: the cost of sampling*: Likewise, suppose we want to draw a sample from the model. The naive way to do this is to sample some value $\mathrm{u} \sim U(0, 1)$, then iterate through the table, adding up the probability values until they exceed u and return the outcome corresponding to that position in the table. This requires reading through the whole table in the worst case, so it has the same exponential cost as the other operations.

The problem with the table-based approach is that we are explicitly modeling every possible kind of interaction between every possible subset of variables. The probability distributions we encounter in real tasks are much simpler than this. Usually, most variables influence each other only indirectly.

For example, consider modeling the finishing times of a team in a relay race. Suppose the team consists of three runners: Alice, Bob and Carol. At the start of the race, Alice carries a baton and begins running around a track. After completing her lap around the track, she hands the baton to Bob. Bob then runs his own lap and hands the baton to Carol, who runs the final lap. We can model each of their finishing times as a continuous random variable. Alice's finishing time does not depend on anyone else's, since she goes first. Bob's finishing time depends on Alice's, because Bob does not have the opportunity to start his lap until Alice has completed hers. If Alice finishes faster, Bob will finish faster, all else being

equal. Finally, Carol's finishing time depends on both her teammates. If Alice is slow, Bob will probably finish late too. As a consequence, Carol will have quite a late starting time and thus is likely to have a late finishing time as well. However, Carol's finishing time depends only *indirectly* on Alice's finishing time via Bob's. If we already know Bob's finishing time, we will not be able to estimate Carol's finishing time better by finding out what Alice's finishing time was. This means we can model the relay race using only two interactions: Alice's effect on Bob and Bob's effect on Carol. We can omit the third, indirect interaction between Alice and Carol from our model.

Structured probabilistic models provide a formal framework for modeling only direct interactions between random variables. This allows the models to have significantly fewer parameters and therefore be estimated reliably from less data. These smaller models also have dramatically reduced computational cost in terms of storing the model, performing inference in the model, and drawing samples from the model.

16.2 Using Graphs to Describe Model Structure

Structured probabilistic models use graphs (in the graph theory sense of "nodes" or "vertices" connected by edges) to represent interactions between random variables. Each node represents a random variable. Each edge represents a direct interaction. These direct interactions imply other, indirect interactions, but only the direct interactions need to be explicitly modeled.

There is more than one way to describe the interactions in a probability distribution using a graph. In the following sections we describe some of the most popular and useful approaches. Graphical models can be largely divided into two categories: models based on directed acyclic graphs, and models based on undirected graphs.

16.2.1 Directed Models

One kind of structured probabilistic model is the **directed graphical model**, otherwise known as the **belief network** or **Bayesian network**[2] (Pearl, 1985).

Directed graphical models are called "directed" because their edges are directed,

[2] Judea Pearl suggested using the term "Bayesian network" when one wishes to "emphasize the judgmental" nature of the values computed by the network, i.e. to highlight that they usually represent degrees of belief rather than frequencies of events.

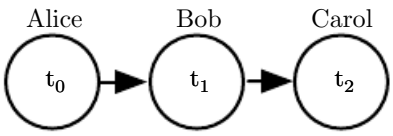


Figure 16.2: A directed graphical model depicting the relay race example. Alice's finishing time t_0 influences Bob's finishing time t_1, because Bob does not get to start running until Alice finishes. Likewise, Carol only gets to start running after Bob finishes, so Bob's finishing time t_1 directly influences Carol's finishing time t_2.

that is, they point from one vertex to another. This direction is represented in the drawing with an arrow. The direction of the arrow indicates which variable's probability distribution is defined in terms of the other's. Drawing an arrow from a to b means that we define the probability distribution over b via a conditional distribution, with a as one of the variables on the right side of the conditioning bar. In other words, the distribution over b depends on the value of a.

Continuing with the relay race example from section 16.1, suppose we name Alice's finishing time t_0, Bob's finishing time t_1, and Carol's finishing time t_2. As we saw earlier, our estimate of t_1 depends on t_0. Our estimate of t_2 depends directly on t_1 but only indirectly on t_0. We can draw this relationship in a directed graphical model, illustrated in figure 16.2.

Formally, a directed graphical model defined on variables $\mathbf{x}$ is defined by a directed acyclic graph $\mathcal{G}$ whose vertices are the random variables in the model, and a set of **local conditional probability distributions** $p(x_i \mid Pa_{\mathcal{G}}(x_i))$ where $Pa_{\mathcal{G}}(x_i)$ gives the parents of x_i in $\mathcal{G}$. The probability distribution over $\mathbf{x}$ is given by

$$p(\mathbf{x}) = \Pi_i p(x_i \mid Pa_{\mathcal{G}}(x_i)). \tag{16.1}$$

In our relay race example, this means that, using the graph drawn in figure 16.2,

$$p(t_0, t_1, t_2) = p(t_0)p(t_1 \mid t_0)p(t_2 \mid t_1). \tag{16.2}$$

This is our first time seeing a structured probabilistic model in action. We can examine the cost of using it, in order to observe how structured modeling has many advantages relative to unstructured modeling.

Suppose we represented time by discretizing time ranging from minute 0 to minute 10 into 6 second chunks. This would make t_0, t_1 and t_2 each be a discrete variable with 100 possible values. If we attempted to represent $p(t_0, t_1, t_2)$ with a table, it would need to store 999,999 values (100 values of t_0 $\times$ 100 values of t_1 $\times$ 100 values of t_2, minus 1, since the probability of one of the configurations is made

redundant by the constraint that the sum of the probabilities be 1). If instead, we only make a table for each of the conditional probability distributions, then the distribution over t_0 requires 99 values, the table defining t_1 given t_0 requires 9900 values, and so does the table defining t_2 given t_1. This comes to a total of 19,899 values. This means that using the directed graphical model reduced our number of parameters by a factor of more than 50!

In general, to model n discrete variables each having k values, the cost of the single table approach scales like $O(k^n)$, as we have observed before. Now suppose we build a directed graphical model over these variables. If m is the maximum number of variables appearing (on either side of the conditioning bar) in a single conditional probability distribution, then the cost of the tables for the directed model scales like $O(k^m)$. As long as we can design a model such that $m << n$, we get very dramatic savings.

In other words, so long as each variable has few parents in the graph, the distribution can be represented with very few parameters. Some restrictions on the graph structure, such as requiring it to be a tree, can also guarantee that operations like computing marginal or conditional distributions over subsets of variables are efficient.

It is important to realize what kinds of information can and cannot be encoded in the graph. The graph encodes only simplifying assumptions about which variables are conditionally independent from each other. It is also possible to make other kinds of simplifying assumptions. For example, suppose we assume Bob always runs the same regardless of how Alice performed. (In reality, Alice's performance probably influences Bob's performance—depending on Bob's personality, if Alice runs especially fast in a given race, this might encourage Bob to push hard and match her exceptional performance, or it might make him overconfident and lazy). Then the only effect Alice has on Bob's finishing time is that we must add Alice's finishing time to the total amount of time we think Bob needs to run. This observation allows us to define a model with $O(k)$ parameters instead of $O(k^2)$. However, note that t_0 and t_1 are still directly dependent with this assumption, because t_1 represents the absolute time at which Bob finishes, not the total time he himself spends running. This means our graph must still contain an arrow from t_0 to t_1. The assumption that Bob's personal running time is independent from all other factors cannot be encoded in a graph over t_0, t_1, and t_2. Instead, we encode this information in the definition of the conditional distribution itself. The conditional distribution is no longer a $k \times k - 1$ element table indexed by t_0 and t_1 but is now a slightly more complicated formula using only $k - 1$ parameters. The directed graphical model syntax does not place any constraint on how we define

our conditional distributions. It only defines which variables they are allowed to take in as arguments.

16.2.2 Undirected Models

Directed graphical models give us one language for describing structured probabilistic models. Another popular language is that of **undirected models**, otherwise known as **Markov random fields** (MRFs) or **Markov networks** (Kindermann, 1980). As their name implies, undirected models use graphs whose edges are undirected.

Directed models are most naturally applicable to situations where there is a clear reason to draw each arrow in one particular direction. Often these are situations where we understand the causality and the causality only flows in one direction. One such situation is the relay race example. Earlier runners affect the finishing times of later runners; later runners do not affect the finishing times of earlier runners.

Not all situations we might want to model have such a clear direction to their interactions. When the interactions seem to have no intrinsic direction, or to operate in both directions, it may be more appropriate to use an undirected model.

As an example of such a situation, suppose we want to model a distribution over three binary variables: whether or not you are sick, whether or not your coworker is sick, and whether or not your roommate is sick. As in the relay race example, we can make simplifying assumptions about the kinds of interactions that take place. Assuming that your coworker and your roommate do not know each other, it is very unlikely that one of them will give the other an infection such as a cold directly. This event can be seen as so rare that it is acceptable not to model it. However, it is reasonably likely that either of them could give you a cold, and that you could pass it on to the other. We can model the indirect transmission of a cold from your coworker to your roommate by modeling the transmission of the cold from your coworker to you and the transmission of the cold from you to your roommate.

In this case, it is just as easy for you to cause your roommate to get sick as it is for your roommate to make you sick, so there is not a clean, uni-directional narrative on which to base the model. This motivates using an undirected model. As with directed models, if two nodes in an undirected model are connected by an edge, then the random variables corresponding to those nodes interact with each other directly. Unlike directed models, the edge in an undirected model has no arrow, and is not associated with a conditional probability distribution.

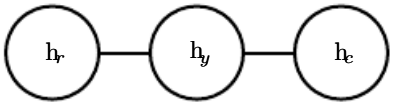


Figure 16.3: An undirected graph representing how your roommate's health h_r, your health h_y, and your work colleague's health h_c affect each other. You and your roommate might infect each other with a cold, and you and your work colleague might do the same, but assuming that your roommate and your colleague do not know each other, they can only infect each other indirectly via you.

We denote the random variable representing your health as h_y, the random variable representing your roommate's health as h_r, and the random variable representing your colleague's health as h_c. See figure 16.3 for a drawing of the graph representing this scenario.

Formally, an undirected graphical model is a structured probabilistic model defined on an undirected graph $\mathcal{G}$. For each clique $\mathcal{C}$ in the graph,[3] a **factor** $\phi(\mathcal{C})$ (also called a **clique potential**) measures the affinity of the variables in that clique for being in each of their possible joint states. The factors are constrained to be non-negative. Together they define an **unnormalized probability distribution**

$$\tilde{p}(\mathbf{x}) = \Pi_{\mathcal{C} \in \mathcal{G}} \phi(\mathcal{C}). \tag{16.3}$$

The unnormalized probability distribution is efficient to work with so long as all the cliques are small. It encodes the idea that states with higher affinity are more likely. However, unlike in a Bayesian network, there is little structure to the definition of the cliques, so there is nothing to guarantee that multiplying them together will yield a valid probability distribution. See figure 16.4 for an example of reading factorization information from an undirected graph.

Our example of the cold spreading between you, your roommate, and your colleague contains two cliques. One clique contains h_y and h_c. The factor for this clique can be defined by a table, and might have values resembling these:

	$h_y = 0$	$h_y = 1$
$h_c = 0$	2	1
$h_c = 1$	1	10

[3] A clique of the graph is a subset of nodes that are all connected to each other by an edge of the graph.

A state of 1 indicates good health, while a state of 0 indicates poor health (having been infected with a cold). Both of you are usually healthy, so the corresponding state has the highest affinity. The state where only one of you is sick has the lowest affinity, because this is a rare state. The state where both of you are sick (because one of you has infected the other) is a higher affinity state, though still not as common as the state where both are healthy.

To complete the model, we would need to also define a similar factor for the clique containing h_y and h_r.

16.2.3 The Partition Function

While the unnormalized probability distribution is guaranteed to be non-negative everywhere, it is not guaranteed to sum or integrate to 1. To obtain a valid probability distribution, we must use the corresponding normalized probability distribution:[4]

$$p(\mathbf{x}) = \frac{1}{Z}\tilde{p}(\mathbf{x}) \tag{16.4}$$

where Z is the value that results in the probability distribution summing or integrating to 1:

$$Z = \int \tilde{p}(\mathbf{x}) d\mathbf{x}. \tag{16.5}$$

You can think of Z as a constant when the ϕ functions are held constant. Note that if the ϕ functions have parameters, then Z is a function of those parameters. It is common in the literature to write Z with its arguments omitted to save space. The normalizing constant Z is known as the **partition function**, a term borrowed from statistical physics.

Since Z is an integral or sum over all possible joint assignments of the state $\mathbf{x}$ it is often intractable to compute. In order to be able to obtain the normalized probability distribution of an undirected model, the model structure and the definitions of the ϕ functions must be conducive to computing Z efficiently. In the context of deep learning, Z is usually intractable. Due to the intractability of computing Z exactly, we must resort to approximations. Such approximate algorithms are the topic of chapter 18.

One important consideration to keep in mind when designing undirected models is that it is possible to specify the factors in such a way that Z does not exist. This happens if some of the variables in the model are continuous and the integral

[4] A distribution defined by normalizing a product of clique potentials is also called a **Gibbs distribution**.

of $\tilde{p}$ over their domain diverges. For example, suppose we want to model a single scalar variable $\mathrm{x} \in \mathbb{R}$ with a single clique potential $\phi(x) = x^2$. In this case,

$$Z = \int x^2 dx. \tag{16.6}$$

Since this integral diverges, there is no probability distribution corresponding to this choice of $\phi(x)$. Sometimes the choice of some parameter of the ϕ functions determines whether the probability distribution is defined. For example, for $\phi(x; \beta) = \exp\left(-\beta x^2\right)$, the β parameter determines whether Z exists. Positive β results in a Gaussian distribution over x but all other values of β make ϕ impossible to normalize.

One key difference between directed modeling and undirected modeling is that directed models are defined directly in terms of probability distributions from the start, while undirected models are defined more loosely by ϕ functions that are then converted into probability distributions. This changes the intuitions one must develop in order to work with these models. One key idea to keep in mind while working with undirected models is that the domain of each of the variables has dramatic effect on the kind of probability distribution that a given set of ϕ functions corresponds to. For example, consider an n-dimensional vector-valued random variable $\mathbf{x}$ and an undirected model parametrized by a vector of biases $\boldsymbol{b}$. Suppose we have one clique for each element of $\mathbf{x}$, $\phi^{(i)}(\mathrm{x}_i) = \exp(b_i \mathrm{x}_i)$. What kind of probability distribution does this result in? The answer is that we do not have enough information, because we have not yet specified the domain of $\mathbf{x}$. If $\mathbf{x} \in \mathbb{R}^n$, then the integral defining Z diverges and no probability distribution exists. If $\mathbf{x} \in \{0, 1\}^n$, then $p(\mathbf{x})$ factorizes into n independent distributions, with $p(\mathrm{x}_i = 1) = \mathrm{sigmoid}(b_i)$. If the domain of $\mathbf{x}$ is the set of elementary basis vectors $(\{[1, 0, \ldots, 0], [0, 1, \ldots, 0], \ldots, [0, 0, \ldots, 1]\})$ then $p(\mathbf{x}) = \mathrm{softmax}(\boldsymbol{b})$, so a large value of b_i actually reduces $p(\mathrm{x}_j = 1)$ for $j \neq i$. Often, it is possible to leverage the effect of a carefully chosen domain of a variable in order to obtain complicated behavior from a relatively simple set of ϕ functions. We will explore a practical application of this idea later, in section 20.6.

16.2.4 Energy-Based Models

Many interesting theoretical results about undirected models depend on the assumption that $\forall \mathbf{x}, \tilde{p}(\mathbf{x}) > 0$. A convenient way to enforce this condition is to use an **energy-based model** (EBM) where

$$\tilde{p}(\mathbf{x}) = \exp(-E(\mathbf{x})) \tag{16.7}$$

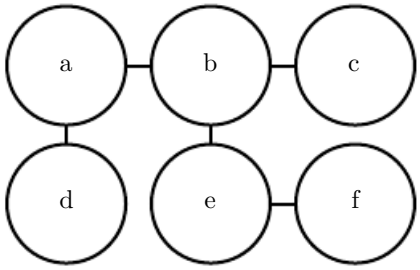


Figure 16.4: This graph implies that $p(\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}, \mathrm{e}, \mathrm{f})$ can be written as $\frac{1}{Z}\, \phi_{\mathrm{a,b}}(\mathrm{a}, \mathrm{b})\phi_{\mathrm{b,c}}(\mathrm{b}, \mathrm{c})\phi_{\mathrm{a,d}}(\mathrm{a}, \mathrm{d})\phi_{\mathrm{b,e}}(\mathrm{b}, \mathrm{e})\, \phi_{\mathrm{e,f}}(\mathrm{e}, \mathrm{f})$ for an appropriate choice of the ϕ functions.

and $E(\mathbf{x})$ is known as the **energy function**. Because $\exp(z)$ is positive for all z, this guarantees that no energy function will result in a probability of zero for any state $\mathbf{x}$. Being completely free to choose the energy function makes learning simpler. If we learned the clique potentials directly, we would need to use constrained optimization to arbitrarily impose some specific minimal probability value. By learning the energy function, we can use unconstrained optimization.[5] The probabilities in an energy-based model can approach arbitrarily close to zero but never reach it.

Any distribution of the form given by equation 16.7 is an example of a **Boltzmann distribution**. For this reason, many energy-based models are called **Boltzmann machines** (Fahlman *et al.*, 1983; Ackley *et al.*, 1985; Hinton *et al.*, 1984; Hinton and Sejnowski, 1986). There is no accepted guideline for when to call a model an energy-based model and when to call it a Boltzmann machine. The term Boltzmann machine was first introduced to describe a model with exclusively binary variables, but today many models such as the mean-covariance restricted Boltzmann machine incorporate real-valued variables as well. While Boltzmann machines were originally defined to encompass both models with and without latent variables, the term Boltzmann machine is today most often used to designate models with latent variables, while Boltzmann machines without latent variables are more often called Markov random fields or log-linear models.

Cliques in an undirected graph correspond to factors of the unnormalized probability function. Because $\exp(a)\exp(b) = \exp(a+b)$, this means that different cliques in the undirected graph correspond to the different terms of the energy function. In other words, an energy-based model is just a special kind of Markov network: the exponentiation makes each term in the energy function correspond to a factor for a different clique. See figure 16.5 for an example of how to read the

[5] For some models, we may still need to use constrained optimization to make sure Z exists.

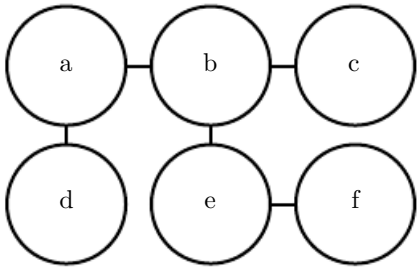
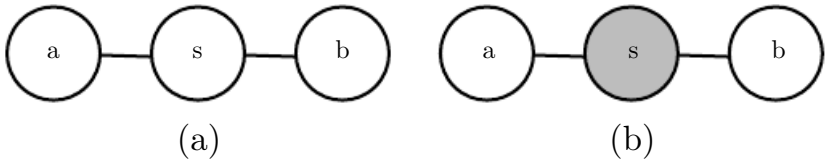


Figure 16.5: This graph implies that $E(\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}, \mathrm{e}, \mathrm{f})$ can be written as $E_{\mathrm{a,b}}(\mathrm{a}, \mathrm{b}) + E_{\mathrm{b,c}}(\mathrm{b}, \mathrm{c}) + E_{\mathrm{a,d}}(\mathrm{a}, \mathrm{d}) + E_{\mathrm{b,e}}(\mathrm{b}, \mathrm{e}) + E_{\mathrm{e,f}}(\mathrm{e}, \mathrm{f})$ for an appropriate choice of the per-clique energy functions. Note that we can obtain the ϕ functions in figure 16.4 by setting each ϕ to the exponential of the corresponding negative energy, e.g., $\phi_{\mathrm{a,b}}(\mathrm{a}, \mathrm{b}) = \exp(-E(\mathrm{a}, \mathrm{b}))$.

form of the energy function from an undirected graph structure. One can view an energy-based model with multiple terms in its energy function as being a **product of experts** (Hinton, 1999). Each term in the energy function corresponds to another factor in the probability distribution. Each term of the energy function can be thought of as an "expert" that determines whether a particular soft constraint is satisfied. Each expert may enforce only one constraint that concerns only a low-dimensional projection of the random variables, but when combined by multiplication of probabilities, the experts together enforce a complicated high-dimensional constraint.

One part of the definition of an energy-based model serves no functional purpose from a machine learning point of view: the $-$ sign in equation 16.7. This $-$ sign could be incorporated into the definition of E. For many choices of the function E, the learning algorithm is free to determine the sign of the energy anyway. The $-$ sign is present primarily to preserve compatibility between the machine learning literature and the physics literature. Many advances in probabilistic modeling were originally developed by statistical physicists, for whom E refers to actual, physical energy and does not have arbitrary sign. Terminology such as "energy" and "partition function" remains associated with these techniques, even though their mathematical applicability is broader than the physics context in which they were developed. Some machine learning researchers (e.g., Smolensky (1986), who referred to negative energy as **harmony**) have chosen to emit the negation, but this is not the standard convention.

Many algorithms that operate on probabilistic models do not need to compute $p_{\mathrm{model}}(\boldsymbol{x})$ but only $\log \tilde{p}_{\mathrm{model}}(\boldsymbol{x})$. For energy-based models with latent variables $\boldsymbol{h}$, these algorithms are sometimes phrased in terms of the negative of this quantity,



(a) (b)

Figure 16.6: (a) The path between random variable a and random variable b through s is active, because s is not observed. This means that a and b are not separated. (b) Here s is shaded in, to indicate that it is observed. Because the only path between a and b is through s, and that path is inactive, we can conclude that a and b are separated given s.

called the **free energy**:

$$\mathcal{F}(\boldsymbol{x}) = -\log \sum_{\boldsymbol{h}} \exp\left(-E(\boldsymbol{x}, \boldsymbol{h})\right). \tag{16.8}$$

In this book, we usually prefer the more general $\log \tilde{p}_{\text{model}}(\boldsymbol{x})$ formulation.

16.2.5 Separation and D-Separation

The edges in a graphical model tell us which variables directly interact. We often need to know which variables *indirectly* interact. Some of these indirect interactions can be enabled or disabled by observing other variables. More formally, we would like to know which subsets of variables are conditionally independent from each other, given the values of other subsets of variables.

Identifying the conditional independences in a graph is very simple in the case of undirected models. In this case, conditional independence implied by the graph is called **separation**. We say that a set of variables $\mathbb{A}$ is **separated** from another set of variables $\mathbb{B}$ given a third set of variables $\mathbb{S}$ if the graph structure implies that $\mathbb{A}$ is independent from $\mathbb{B}$ given $\mathbb{S}$. If two variables a and b are connected by a path involving only unobserved variables, then those variables are not separated. If no path exists between them, or all paths contain an observed variable, then they are separated. We refer to paths involving only unobserved variables as "active" and paths including an observed variable as "inactive."

When we draw a graph, we can indicate observed variables by shading them in. See figure 16.6 for a depiction of how active and inactive paths in an undirected model look when drawn in this way. See figure 16.7 for an example of reading separation from an undirected graph.

Similar concepts apply to directed models, except that in the context of directed models, these concepts are referred to as **d-separation**. The "d" stands for "dependence." D-separation for directed graphs is defined the same as separation

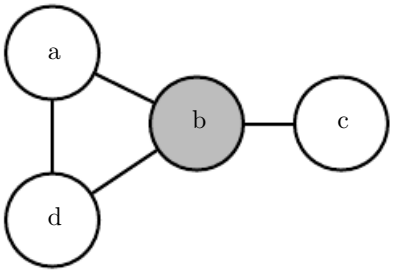


Figure 16.7: An example of reading separation properties from an undirected graph. Here b is shaded to indicate that it is observed. Because observing b blocks the only path from a to c, we say that a and c are separated from each other given b. The observation of b also blocks one path between a and d, but there is a second, active path between them. Therefore, a and d are not separated given b.

for undirected graphs: We say that a set of variables $\mathbb{A}$ is d-separated from another set of variables $\mathbb{B}$ given a third set of variables $\mathbb{S}$ if the graph structure implies that $\mathbb{A}$ is independent from $\mathbb{B}$ given $\mathbb{S}$.

As with undirected models, we can examine the independences implied by the graph by looking at what active paths exist in the graph. As before, two variables are dependent if there is an active path between them, and d-separated if no such path exists. In directed nets, determining whether a path is active is somewhat more complicated. See figure 16.8 for a guide to identifying active paths in a directed model. See figure 16.9 for an example of reading some properties from a graph.

It is important to remember that separation and d-separation tell us only about those conditional independences *that are implied by the graph.* There is no requirement that the graph imply all independences that are present. In particular, it is always legitimate to use the complete graph (the graph with all possible edges) to represent any distribution. In fact, some distributions contain independences that are not possible to represent with existing graphical notation. **Context-specific independences** are independences that are present dependent on the value of some variables in the network. For example, consider a model of three binary variables: a, b and c. Suppose that when a is 0, b and c are independent, but when a is 1, b is deterministically equal to c. Encoding the behavior when $a = 1$ requires an edge connecting b and c. The graph then fails to indicate that b and c are independent when $a = 0$.

In general, a graph will never imply that an independence exists when it does not. However, a graph may fail to encode an independence.

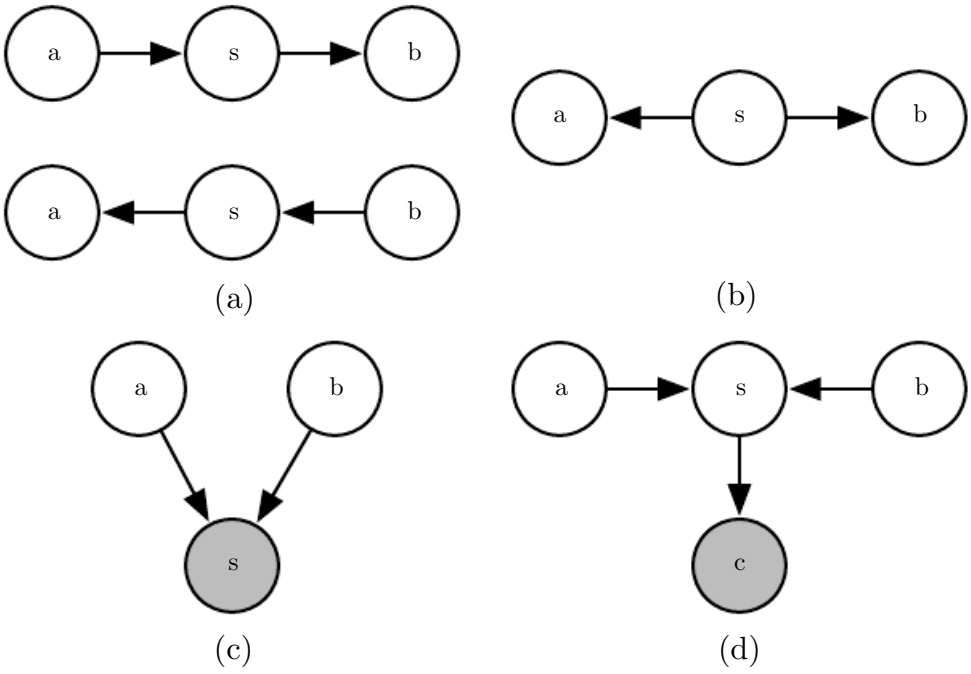


Figure 16.8: All of the kinds of active paths of length two that can exist between random variables a and b. *(a)*Any path with arrows proceeding directly froma to b or vice versa. This kind of path becomes blocked if s is observed. We have already seen this kind of path in the relay race example. *(b)*a and b are connected by a *common cause* s. For example, suppose s is a variable indicating whether or not there is a hurricane and a and b measure the wind speed at two different nearby weather monitoring outposts. If we observe very high winds at station a, we might expect to also see high winds at b. This kind of path can be blocked by observing s. If we already know there is a hurricane, we expect to see high winds at b, regardless of what is observed at a. A lower than expected wind at a (for a hurricane) would not change our expectation of winds at b (knowing there is a hurricane). However, if s is not observed, then a and b are dependent, i.e., the path is active. *(c)*a and b are both parents of s. This is called a **V-structure** or **the collider case**. The V-structure causes a and b to be related by the **explaining away effect**. In this case, the path is actually active when s is observed. For example, suppose s is a variable indicating that your colleague is not at work. The variable a represents her being sick, while b represents her being on vacation. If you observe that she is not at work, you can presume she is probably sick or on vacation, but it is not especially likely that both have happened at the same time. If you find out that she is on vacation, this fact is sufficient to *explain* her absence. You can infer that she is probably not also sick. *(d)*The explaining away effect happens even if any descendant ofs is observed! For example, suppose that c is a variable representing whether you have received a report from your colleague. If you notice that you have not received the report, this increases your estimate of the probability that she is not at work today, which in turn makes it more likely that she is either sick or on vacation. The only way to block a path through a V-structure is to observe none of the descendants of the shared child.

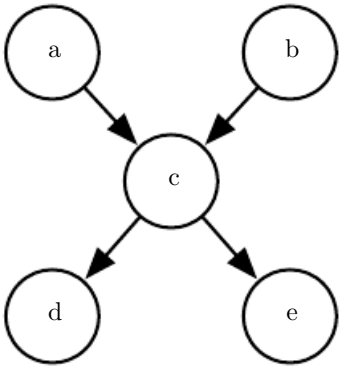


Figure 16.9: From this graph, we can read out several d-separation properties. Examples include:

- a and b are d-separated given the empty set.
- a and e are d-separated given c.
- d and e are d-separated given c.

We can also see that some variables are no longer d-separated when we observe some variables:

- a and b are not d-separated given c.
- a and b are not d-separated given d.

16.2.6 Converting between Undirected and Directed Graphs

We often refer to a specific machine learning model as being undirected or directed. For example, we typically refer to RBMs as undirected and sparse coding as directed. This choice of wording can be somewhat misleading, because no probabilistic model is inherently directed or undirected. Instead, some models are most easily *described* using a directed graph, or most easily described using an undirected graph.

Directed models and undirected models both have their advantages and disadvantages. Neither approach is clearly superior and universally preferred. Instead, we should choose which language to use for each task. This choice will partially depend on which probability distribution we wish to describe. We may choose to use either directed modeling or undirected modeling based on which approach can capture the most independences in the probability distribution or which approach uses the fewest edges to describe the distribution. There are other factors that can affect the decision of which language to use. Even while working with a single probability distribution, we may sometimes switch between different modeling languages. Sometimes a different language becomes more appropriate if we observe a certain subset of variables, or if we wish to perform a different computational task. For example, the directed model description often provides a straightforward approach to efficiently draw samples from the model (described in section 16.3) while the undirected model formulation is often useful for deriving approximate inference procedures (as we will see in chapter 19, where the role of undirected models is highlighted in equation 19.56).

Every probability distribution can be represented by either a directed model or by an undirected model. In the worst case, one can always represent any distribution by using a "complete graph." In the case of a directed model, the complete graph is any directed acyclic graph where we impose some ordering on the random variables, and each variable has all other variables that precede it in the ordering as its ancestors in the graph. For an undirected model, the complete graph is simply a graph containing a single clique encompassing all of the variables. See figure 16.10 for an example.

Of course, the utility of a graphical model is that the graph implies that some variables do not interact directly. The complete graph is not very useful because it does not imply any independences.

When we represent a probability distribution with a graph, we want to choose a graph that implies as many independences as possible, without implying any independences that do not actually exist.

From this point of view, some distributions can be represented more efficiently

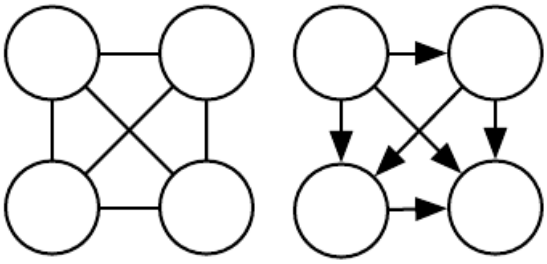


Figure 16.10: Examples of complete graphs, which can describe any probability distribution. Here we show examples with four random variables. *(Left)*The complete undirected graph. In the undirected case, the complete graph is unique. *(Right)*A complete directed graph. In the directed case, there is not a unique complete graph. We choose an ordering of the variables and draw an arc from each variable to every variable that comes after it in the ordering. There are thus a factorial number of complete graphs for every set of random variables. In this example we order the variables from left to right, top to bottom.

using directed models, while other distributions can be represented more efficiently using undirected models. In other words, directed models can encode some independences that undirected models cannot encode, and vice versa.

Directed models are able to use one specific kind of substructure that undirected models cannot represent perfectly. This substructure is called an **immorality**. The structure occurs when two random variables a and b are both parents of a third random variable c, and there is no edge directly connecting a and b in either direction. (The name "immorality" may seem strange; it was coined in the graphical models literature as a joke about unmarried parents.) To convert a directed model with graph $\mathcal{D}$ into an undirected model, we need to create a new graph $\mathcal{U}$. For every pair of variables x and y, we add an undirected edge connecting x and y to $\mathcal{U}$ if there is a directed edge (in either direction) connecting x and y in $\mathcal{D}$ or if x and y are both parents in $\mathcal{D}$ of a third variable z. The resulting $\mathcal{U}$ is known as a **moralized graph**. See figure 16.11 for examples of converting directed models to undirected models via moralization.

Likewise, undirected models can include substructures that no directed model can represent perfectly. Specifically, a directed graph $\mathcal{D}$ cannot capture all of the conditional independences implied by an undirected graph $\mathcal{U}$ if $\mathcal{U}$ contains a **loop** of length greater than three, unless that loop also contains a **chord**. A loop is a sequence of variables connected by undirected edges, with the last variable in the sequence connected back to the first variable in the sequence. A chord is a connection between any two non-consecutive variables in the sequence defining a loop. If $\mathcal{U}$ has loops of length four or greater and does not have chords for these loops, we must add the chords before we can convert it to a directed model. Adding

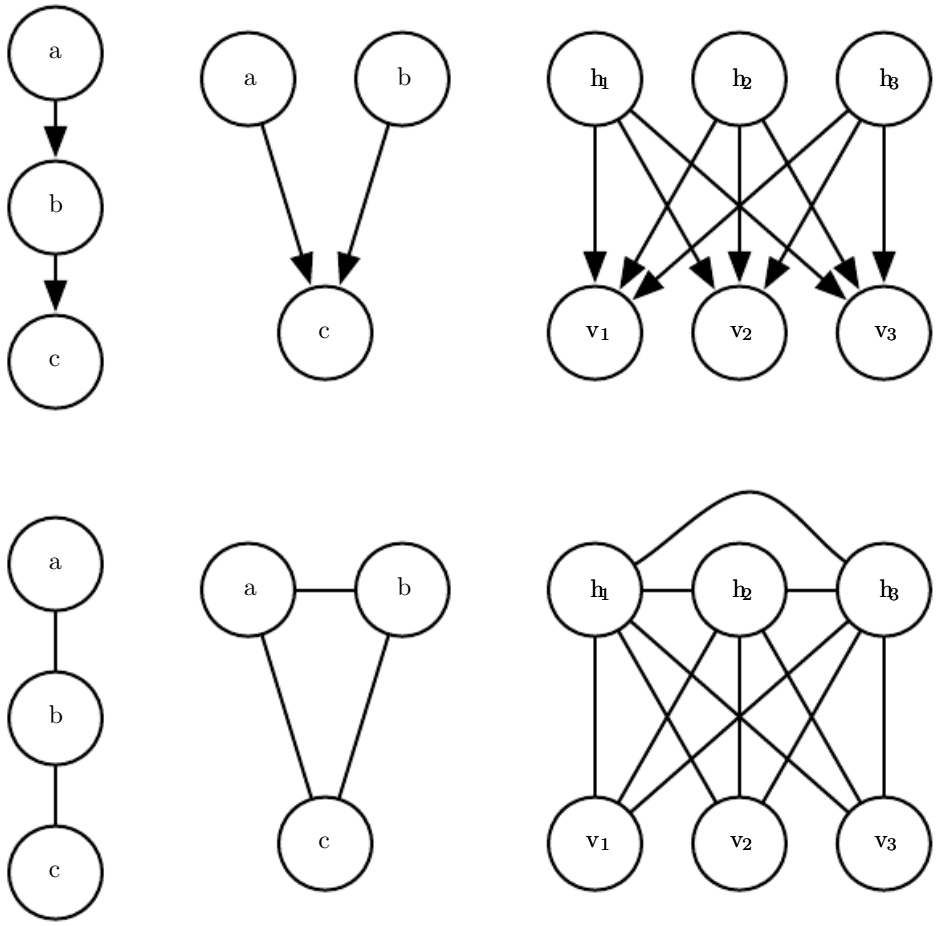


Figure 16.11: Examples of converting directed models (top row) to undirected models (bottom row) by constructing moralized graphs. *(Left)*This simple chain can be converted to a moralized graph merely by replacing its directed edges with undirected edges. The resulting undirected model implies exactly the same set of independences and conditional independences. *(Center)*This graph is the simplest directed model that cannot be converted to an undirected model without losing some independences. This graph consists entirely of a single immorality. Because a and b are parents of c, they are connected by an active path when c is observed. To capture this dependence, the undirected model must include a clique encompassing all three variables. This clique fails to encode the fact that $a \perp b$. *(Right)*In general, moralization may add many edges to the graph, thus losing many implied independences. For example, this sparse coding graph requires adding moralizing edges between every pair of hidden units, thus introducing a quadratic number of new direct dependences.

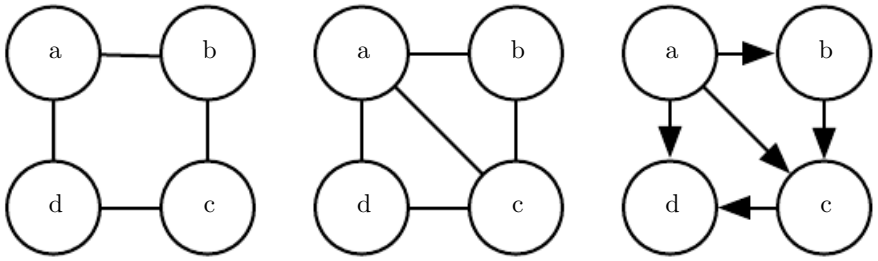


Figure 16.12: Converting an undirected model to a directed model. *(Left)*This undirected model cannot be converted directed to a directed model because it has a loop of length four with no chords. Specifically, the undirected model encodes two different independences that no directed model can capture simultaneously: $\mathrm{a} \perp \mathrm{c} \mid \{\mathrm{b}, \mathrm{d}\}$ and $\mathrm{b} \perp \mathrm{d} \mid \{\mathrm{a}, \mathrm{c}\}$. *(Center)*To convert the undirected model to a directed model, we must triangulate the graph, by ensuring that all loops of greater than length three have a chord. To do so, we can either add an edge connecting a and c or we can add an edge connecting b and d. In this example, we choose to add the edge connectinga and c. *(Right)*To finish the conversion process, we must assign a direction to each edge. When doing so, we must not create any directed cycles. One way to avoid directed cycles is to impose an ordering over the nodes, and always point each edge from the node that comes earlier in the ordering to the node that comes later in the ordering. In this example, we use the variable names to impose alphabetical order.

these chords discards some of the independence information that was encoded in $\mathcal{U}$. The graph formed by adding chords to $\mathcal{U}$ is known as a **chordal** or **triangulated** graph, because all the loops can now be described in terms of smaller, triangular loops. To build a directed graph $\mathcal{D}$ from the chordal graph, we need to also assign directions to the edges. When doing so, we must not create a directed cycle in $\mathcal{D}$, or the result does not define a valid directed probabilistic model. One way to assign directions to the edges in $\mathcal{D}$ is to impose an ordering on the random variables, then point each edge from the node that comes earlier in the ordering to the node that comes later in the ordering. See figure 16.12 for a demonstration.

16.2.7 Factor Graphs

Factor graphs are another way of drawing undirected models that resolve an ambiguity in the graphical representation of standard undirected model syntax. In an undirected model, the scope of every ϕ function must be a *subset* of some clique in the graph. Ambiguity arises because it is not clear if each clique actually has a corresponding factor whose scope encompasses the entire clique—for example, a clique containing three nodes may correspond to a factor over all three nodes, or may correspond to three factors that each contain only a pair of the nodes.

Factor graphs resolve this ambiguity by explicitly representing the scope of each ϕ function. Specifically, a factor graph is a graphical representation of an undirected model that consists of a bipartite undirected graph. Some of the nodes are drawn as circles. These nodes correspond to random variables as in a standard undirected model. The rest of the nodes are drawn as squares. These nodes correspond to the factors ϕ of the unnormalized probability distribution. Variables and factors may be connected with undirected edges. A variable and a factor are connected in the graph if and only if the variable is one of the arguments to the factor in the unnormalized probability distribution. No factor may be connected to another factor in the graph, nor can a variable be connected to a variable. See figure 16.13 for an example of how factor graphs can resolve ambiguity in the interpretation of undirected networks.

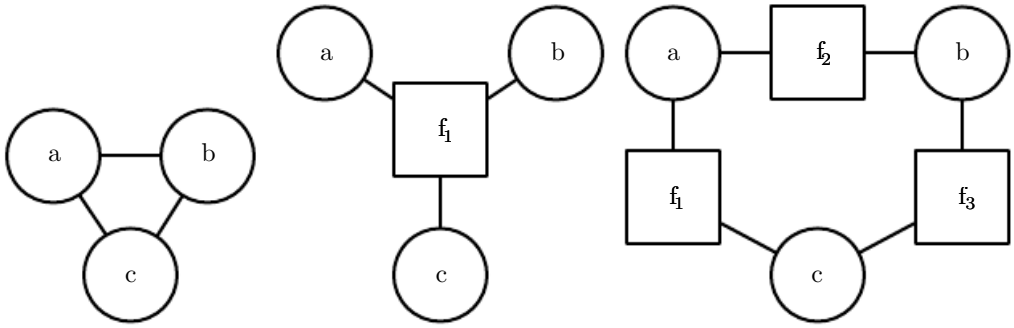


Figure 16.13: An example of how a factor graph can resolve ambiguity in the interpretation of undirected networks. *(Left)*An undirected network with a clique involving three variables: a, b and c. *(Center)*A factor graph corresponding to the same undirected model. This factor graph has one factor over all three variables. *(Right)*Another valid factor graph for the same undirected model. This factor graph has three factors, each over only two variables. Representation, inference, and learning are all asymptotically cheaper in this factor graph than in the factor graph depicted in the center, even though both require the same undirected graph to represent.

16.3 Sampling from Graphical Models

Graphical models also facilitate the task of drawing samples from a model.

One advantage of directed graphical models is that a simple and efficient procedure called **ancestral sampling** can produce a sample from the joint distribution represented by the model.

The basic idea is to sort the variables x_i in the graph into a topological ordering, so that for all i and j, j is greater than i if x_i is a parent of x_j. The variables

can then be sampled in this order. In other words, we first sample $\mathrm{x}_1 \sim P(\mathrm{x}_1)$, then sample $P(\mathrm{x}_2 \mid Pa_{\mathcal{G}}(\mathrm{x}_2))$, and so on, until finally we sample $P(\mathrm{x}_n \mid Pa_{\mathcal{G}}(\mathrm{x}_n))$. So long as each conditional distribution $p(\mathrm{x}_i \mid Pa_{\mathcal{G}}(\mathrm{x}_i))$ is easy to sample from, then the whole model is easy to sample from. The topological sorting operation guarantees that we can read the conditional distributions in equation 16.1 and sample from them in order. Without the topological sorting, we might attempt to sample a variable before its parents are available.

For some graphs, more than one topological ordering is possible. Ancestral sampling may be used with any of these topological orderings.

Ancestral sampling is generally very fast (assuming sampling from each conditional is easy) and convenient.

One drawback to ancestral sampling is that it only applies to directed graphical models. Another drawback is that it does not support every conditional sampling operation. When we wish to sample from a subset of the variables in a directed graphical model, given some other variables, we often require that all the conditioning variables come earlier than the variables to be sampled in the ordered graph. In this case, we can sample from the local conditional probability distributions specified by the model distribution. Otherwise, the conditional distributions we need to sample from are the posterior distributions given the observed variables. These posterior distributions are usually not explicitly specified and parametrized in the model. Inferring these posterior distributions can be costly. In models where this is the case, ancestral sampling is no longer efficient.

Unfortunately, ancestral sampling is applicable only to directed models. We can sample from undirected models by converting them to directed models, but this often requires solving intractable inference problems (to determine the marginal distribution over the root nodes of the new directed graph) or requires introducing so many edges that the resulting directed model becomes intractable. Sampling from an undirected model without first converting it to a directed model seems to require resolving cyclical dependencies. Every variable interacts with every other variable, so there is no clear beginning point for the sampling process. Unfortunately, drawing samples from an undirected graphical model is an expensive, multi-pass process. The conceptually simplest approach is **Gibbs sampling**. Suppose we have a graphical model over an n-dimensional vector of random variables $\mathbf{x}$. We iteratively visit each variable x_i and draw a sample conditioned on all of the other variables, from $p(\mathrm{x}_i \mid \mathrm{x}_{-i})$. Due to the separation properties of the graphical model, we can equivalently condition on only the neighbors of x_i. Unfortunately, after we have made one pass through the graphical model and sampled all n variables, we still do not have a fair sample from $p(\mathbf{x})$. Instead, we must repeat the

process and resample all n variables using the updated values of their neighbors. Asymptotically, after many repetitions, this process converges to sampling from the correct distribution. It can be difficult to determine when the samples have reached a sufficiently accurate approximation of the desired distribution. Sampling techniques for undirected models are an advanced topic, covered in more detail in chapter 17.

16.4 Advantages of Structured Modeling

The primary advantage of using structured probabilistic models is that they allow us to dramatically reduce the cost of representing probability distributions as well as learning and inference. Sampling is also accelerated in the case of directed models, while the situation can be complicated with undirected models. The primary mechanism that allows all of these operations to use less runtime and memory is choosing to not model certain interactions. Graphical models convey information by leaving edges out. Anywhere there is not an edge, the model specifies the assumption that we do not need to model a direct interaction.

A less quantifiable benefit of using structured probabilistic models is that they allow us to explicitly separate representation of knowledge from learning of knowledge or inference given existing knowledge. This makes our models easier to develop and debug. We can design, analyze, and evaluate learning algorithms and inference algorithms that are applicable to broad classes of graphs. Independently, we can design models that capture the relationships we believe are important in our data. We can then combine these different algorithms and structures and obtain a Cartesian product of different possibilities. It would be much more difficult to design end-to-end algorithms for every possible situation.

16.5 Learning about Dependencies

A good generative model needs to accurately capture the distribution over the observed or "visible" variables $\mathbf{v}$. Often the different elements of $\mathbf{v}$ are highly dependent on each other. In the context of deep learning, the approach most commonly used to model these dependencies is to introduce several latent or "hidden" variables, $\mathbf{h}$. The model can then capture dependencies between any pair of variables v_i and v_j indirectly, via direct dependencies between v_i and $\mathbf{h}$, and direct dependencies between $\mathbf{h}$ and v_j.

A good model of $\mathbf{v}$ which did not contain any latent variables would need to

have very large numbers of parents per node in a Bayesian network or very large cliques in a Markov network. Just representing these higher order interactions is costly—both in a computational sense, because the number of parameters that must be stored in memory scales exponentially with the number of members in a clique, but also in a statistical sense, because this exponential number of parameters requires a wealth of data to estimate accurately.

When the model is intended to capture dependencies between visible variables with direct connections, it is usually infeasible to connect all variables, so the graph must be designed to connect those variables that are tightly coupled and omit edges between other variables. An entire field of machine learning called **structure learning** is devoted to this problem For a good reference on structure learning, see (Koller and Friedman, 2009). Most structure learning techniques are a form of greedy search. A structure is proposed, a model with that structure is trained, then given a score. The score rewards high training set accuracy and penalizes model complexity. Candidate structures with a small number of edges added or removed are then proposed as the next step of the search. The search proceeds to a new structure that is expected to increase the score.

Using latent variables instead of adaptive structure avoids the need to perform discrete searches and multiple rounds of training. A fixed structure over visible and hidden variables can use direct interactions between visible and hidden units to impose indirect interactions between visible units. Using simple parameter learning techniques we can learn a model with a fixed structure that imputes the right structure on the marginal $p(\mathbf{v})$.

Latent variables have advantages beyond their role in efficiently capturing $p(\mathbf{v})$. The new variables $\mathbf{h}$ also provide an alternative representation for $\mathbf{v}$. For example, as discussed in section 3.9.6, the mixture of Gaussians model learns a latent variable that corresponds to which category of examples the input was drawn from. This means that the latent variable in a mixture of Gaussians model can be used to do classification. In chapter 14 we saw how simple probabilistic models like sparse coding learn latent variables that can be used as input features for a classifier, or as coordinates along a manifold. Other models can be used in this same way, but deeper models and models with different kinds of interactions can create even richer descriptions of the input. Many approaches accomplish feature learning by learning latent variables. Often, given some model of $\mathbf{v}$ and $\mathbf{h}$, experimental observations show that $\mathbb{E}[\mathbf{h} \mid \mathbf{v}]$ or $\text{argmax}_{\boldsymbol{h}} p(\boldsymbol{h}, \boldsymbol{v})$ is a good feature mapping for $\boldsymbol{v}$.

16.6 Inference and Approximate Inference

One of the main ways we can use a probabilistic model is to ask questions about how variables are related to each other. Given a set of medical tests, we can ask what disease a patient might have. In a latent variable model, we might want to extract features $\mathbb{E}[\mathbf{h} \mid \mathbf{v}]$ describing the observed variables $\mathbf{v}$. Sometimes we need to solve such problems in order to perform other tasks. We often train our models using the principle of maximum likelihood. Because

$$\log p(\boldsymbol{v}) = \mathbb{E}_{\mathbf{h} \sim p(\mathbf{h} \mid \boldsymbol{v})} \left[\log p(\boldsymbol{h}, \boldsymbol{v}) - \log p(\boldsymbol{h} \mid \boldsymbol{v}) \right], \tag{16.9}$$

we often want to compute $p(\mathbf{h} \mid \boldsymbol{v})$ in order to implement a learning rule. All of these are examples of **inference** problems in which we must predict the value of some variables given other variables, or predict the probability distribution over some variables given the value of other variables.

Unfortunately, for most interesting deep models, these inference problems are intractable, even when we use a structured graphical model to simplify them. The graph structure allows us to represent complicated, high-dimensional distributions with a reasonable number of parameters, but the graphs used for deep learning are usually not restrictive enough to also allow efficient inference.

It is straightforward to see that computing the marginal probability of a general graphical model is #P hard. The complexity class #P is a generalization of the complexity class NP. Problems in NP require determining only whether a problem has a solution and finding a solution if one exists. Problems in #P require counting the number of solutions. To construct a worst-case graphical model, imagine that we define a graphical model over the binary variables in a 3-SAT problem. We can impose a uniform distribution over these variables. We can then add one binary latent variable per clause that indicates whether each clause is satisfied. We can then add another latent variable indicating whether all of the clauses are satisfied. This can be done without making a large clique, by building a reduction tree of latent variables, with each node in the tree reporting whether two other variables are satisfied. The leaves of this tree are the variables for each clause. The root of the tree reports whether the entire problem is satisfied. Due to the uniform distribution over the literals, the marginal distribution over the root of the reduction tree specifies what fraction of assignments satisfy the problem. While this is a contrived worst-case example, NP hard graphs commonly arise in practical real-world scenarios.

This motivates the use of approximate inference. In the context of deep learning, this usually refers to variational inference, in which we approximate the

true distribution $p(\mathbf{h} \mid \boldsymbol{v})$ by seeking an approximate distribution $q(\mathbf{h}|\mathbf{v})$ that is as close to the true one as possible. This and other techniques are described in depth in chapter 19.

16.7 The Deep Learning Approach to Structured Probabilistic Models

Deep learning practitioners generally use the same basic computational tools as other machine learning practitioners who work with structured probabilistic models. However, in the context of deep learning, we usually make different design decisions about how to combine these tools, resulting in overall algorithms and models that have a very different flavor from more traditional graphical models.

Deep learning does not always involve especially deep graphical models. In the context of graphical models, we can define the depth of a model in terms of the graphical model graph rather than the computational graph. We can think of a latent variable h_i as being at depth j if the shortest path from h_i to an observed variable is j steps. We usually describe the depth of the model as being the greatest depth of any such h_i. This kind of depth is different from the depth induced by the computational graph. Many generative models used for deep learning have no latent variables or only one layer of latent variables, but use deep computational graphs to define the conditional distributions within a model.

Deep learning essentially always makes use of the idea of distributed representations. Even shallow models used for deep learning purposes (such as pretraining shallow models that will later be composed to form deep ones) nearly always have a single, large layer of latent variables. Deep learning models typically have more latent variables than observed variables. Complicated nonlinear interactions between variables are accomplished via indirect connections that flow through multiple latent variables.

By contrast, traditional graphical models usually contain mostly variables that are at least occasionally observed, even if many of the variables are missing at random from some training examples. Traditional models mostly use higher-order terms and structure learning to capture complicated nonlinear interactions between variables. If there are latent variables, they are usually few in number.

The way that latent variables are designed also differs in deep learning. The deep learning practitioner typically does not intend for the latent variables to take on any specific semantics ahead of time—the training algorithm is free to invent the concepts it needs to model a particular dataset. The latent variables are

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

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

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

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

16.7.1 Example: The Restricted Boltzmann Machine

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

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

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

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

The restrictions on the RBM structure yield the nice properties

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

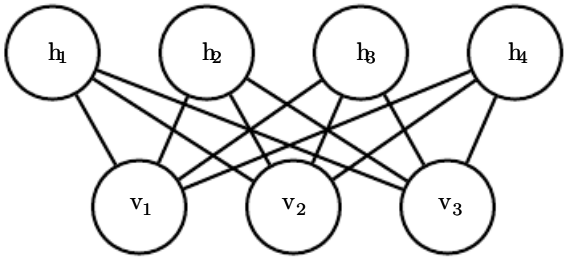


Figure 16.14: An RBM drawn as a Markov network.

and

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

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

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

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

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

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

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

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

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

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

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

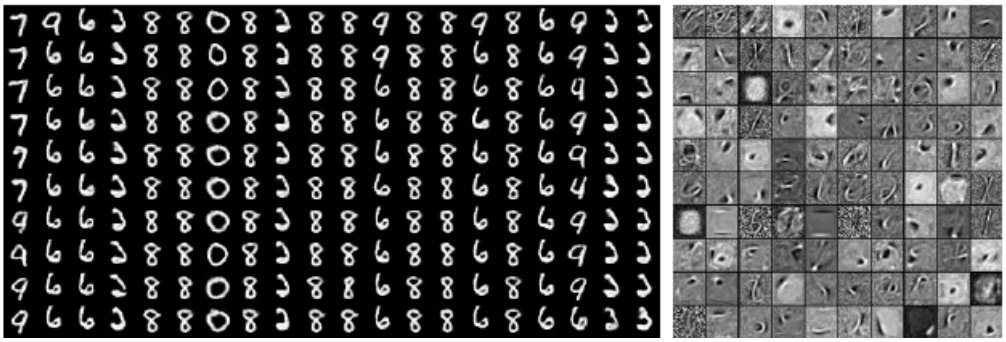


Figure 16.15: Samples from a trained RBM, and its weights. Image reproduced with permission from LISA (2008). *(Left)*Samples from a model trained on MNIST, drawn using Gibbs sampling. Each column is a separate Gibbs sampling process. Each row represents the output of another 1,000 steps of Gibbs sampling.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

Chapter 17

Monte Carlo Methods

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

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

17.1 Sampling and Monte Carlo Methods

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

17.1.1 Why Sampling?

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

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

17.1.2 Basics of Monte Carlo Sampling

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

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

or

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

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

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

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

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

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

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

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

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

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

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

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

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

17.2 Importance Sampling

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

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

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

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

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

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

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

can be transformed into an importance sampling estimator

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

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

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

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

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

The minimum variance occurs when q is

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

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

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

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

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

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

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

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

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

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

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

17.3 Markov Chain Monte Carlo Methods

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

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

of obtaining a sample.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

successive samples.

17.4 Gibbs Sampling

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

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

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

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

17.5 The Challenge of Mixing between Separated Modes

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

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

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

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

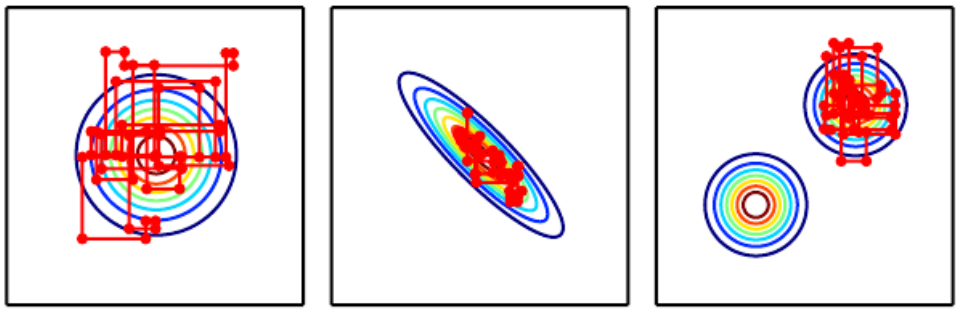


Figure 17.1: Paths followed by Gibbs sampling for three distributions, with the Markov chain initialized at the mode in both cases. *(Left)*A multivariate normal distribution with two independent variables. Gibbs sampling mixes well because the variables are independent. *(Center)*A multivariate normal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

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

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

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

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

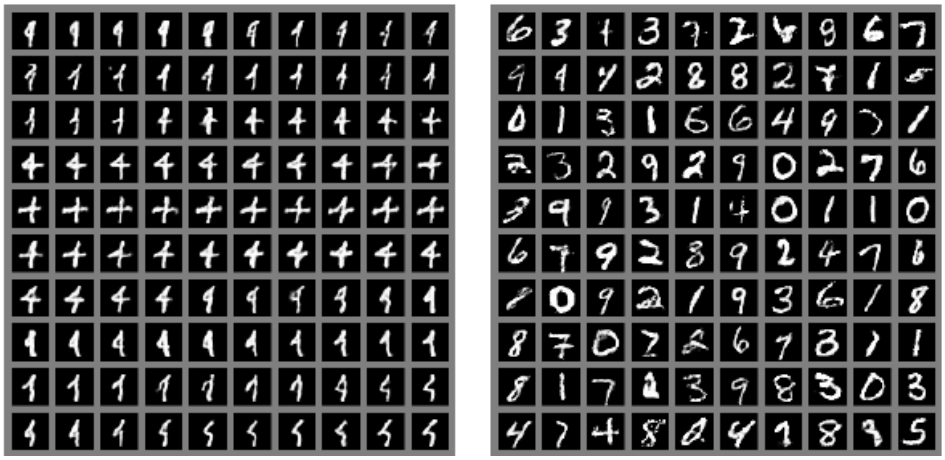


Figure 17.2: An illustration of the slow mixing problem in deep probabilistic models. Each panel should be read left to right, top to bottom. *(Left)*Consecutive samples from Gibbs sampling applied to a deep Boltzmann machine trained on the MNIST dataset. Consecutive samples are similar to each other.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

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

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

17.5.1 Tempering to Mix between Modes

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

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

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

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

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

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

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

17.5.2 Depth May Help Mixing

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

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

Chapter 18

Confronting the Partition Function

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

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

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

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

or

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

This operation is intractable for many interesting models.

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

18.1 The Log-Likelihood Gradient

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

This identity

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

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

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

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

18.2 Stochastic Maximum Likelihood and Contrastive Divergence

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

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

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

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

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

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

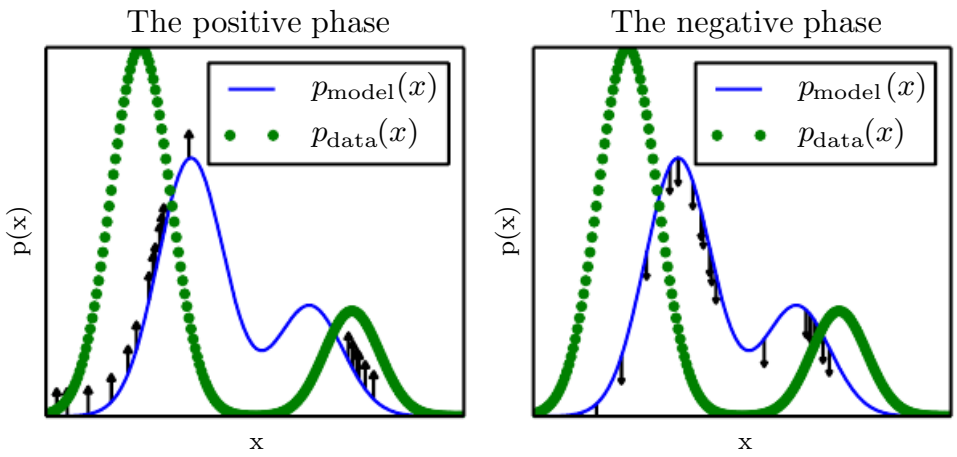


Figure 18.1: The view of algorithm 18.1 as having a "positive phase" and "negative phase." *(Left)*In the positive phase, we sample points from the data distribution, and push up on their unnormalized probability. This means points that are likely in the data get pushed up on more. *(Right)*In the negative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

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

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

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

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

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

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

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

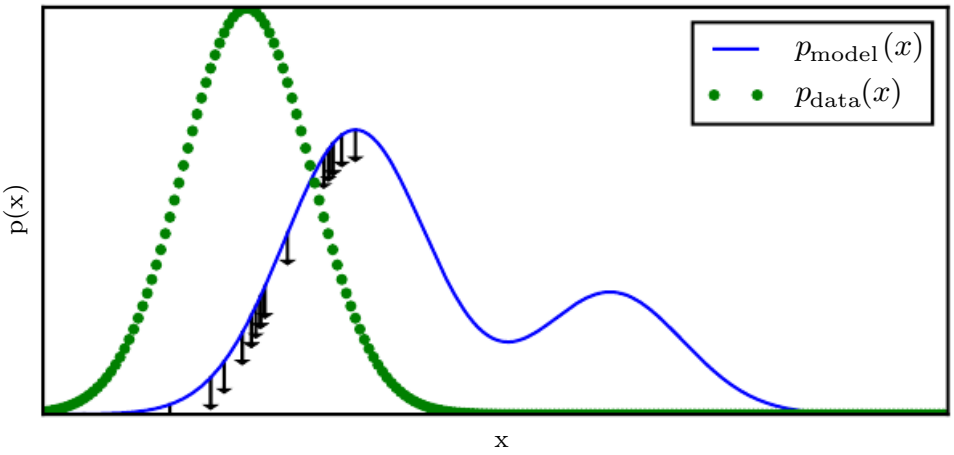


Figure 18.2: An illustration of how the negative phase of contrastive divergence (algorithm 18.2) can fail to suppress spurious modes. A spurious mode is a mode that is present in the model distribution but absent in the data distribution. Because contrastive divergence initializes its Markov chains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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

incompatible with bound-based positive phase methods.

18.3 Pseudolikelihood

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

18.4 Score Matching and Ratio Matching

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

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

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

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

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

equivalent to minimizing the expected value of

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

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

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

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

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

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

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

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

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

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

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

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

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

18.5 Denoising Score Matching

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

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

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

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

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

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

18.6 Noise-Contrastive Estimation

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

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

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

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

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

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

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

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

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

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

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

and

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

18.7 Estimating the Partition Function

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

18.7.1 Annealed Importance Sampling

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- end

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

18.7.2 Bridge Sampling

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

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

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

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

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

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

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

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

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

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

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

Chapter 19

Approximate Inference

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

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

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

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

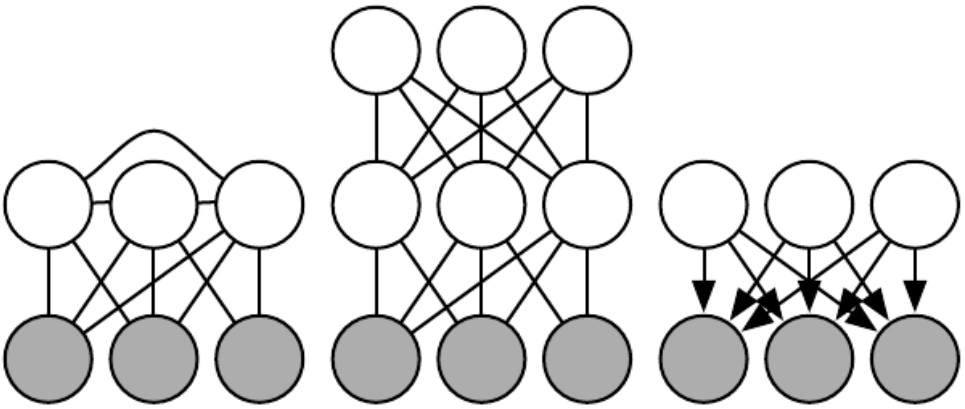


Figure 19.1: Intractable inference problems in deep learning are usually the result of interactions between latent variables in a structured graphical model. These can be due to edges directly connecting one latent variable to another, or due to longer paths that are activated when the child of a V-structure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

19.1 Inference as Optimization

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

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

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

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

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

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

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

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

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

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

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

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

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

19.2 Expectation Maximization

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

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

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

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

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

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

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

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

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

19.3 MAP Inference and Sparse Coding

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

of latent variable models, this means computing

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

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

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

Recall from section 19.1 that exact inference consists of maximizing

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

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

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

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

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

which is equivalent to the MAP inference problem

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

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

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

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

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

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

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

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

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

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

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

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

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

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

19.4 Variational Inference and Learning

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

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

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

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

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

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

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

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

19.4.1 Discrete Latent Variables

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

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

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

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

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

criterion.

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

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

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

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

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

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

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

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

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

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

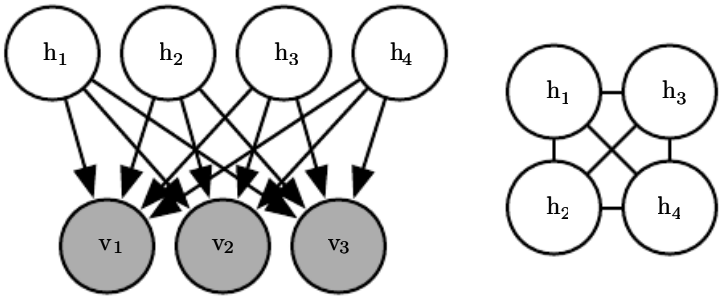


Figure 19.2: The graph structure of a binary sparse coding model with four hidden units. *(Left)*The graph structure of $p(\boldsymbol{h}, \boldsymbol{v})$. Note that the edges are directed, and that every two hidden units are co-parents of every visible unit. *(Right)*The graph structure of $p(\boldsymbol{h}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

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

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

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

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

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

We can make a mean field approximation:

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

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

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

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

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

The evidence lower bound is given by

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

19.4.2 Calculus of Variations

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

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

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

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

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

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

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

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

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

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

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

For continuous values, the expectation is an integral:

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

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

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

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

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

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

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

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

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

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

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

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

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

19.4.3 Continuous Latent Variables

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

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

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

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

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

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

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

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

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

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

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

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

Applying equation 19.56, we find that

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

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

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

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

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

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

19.4.4 Interactions between Learning and Inference

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

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

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

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

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

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

19.5 Learned Approximate Inference

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

19.5.1 Wake-Sleep

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

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

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

19.5.2 Other Forms of Learned Inference

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

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

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

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

Chapter 20

Deep Generative Models

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

20.1 Boltzmann Machines

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

20.2 Restricted Boltzmann Machines

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

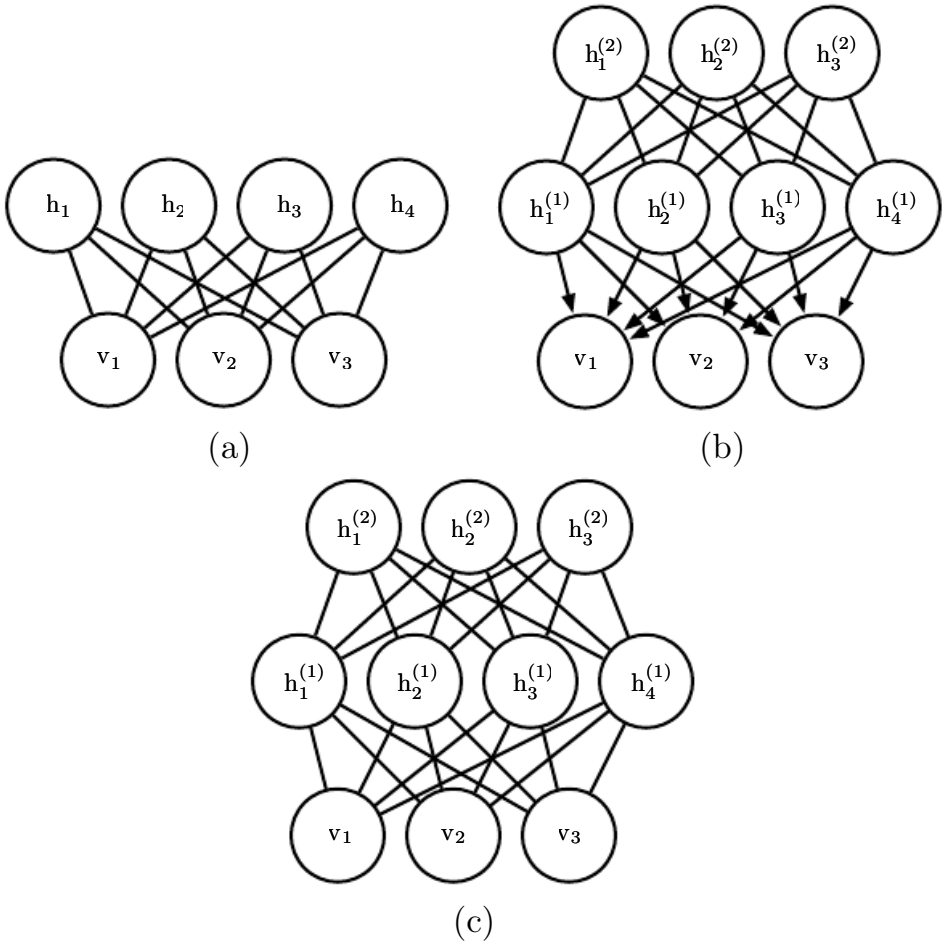


Figure 20.1: Examples of models that may be built with restricted Boltzmann machines. *(a)*The restricted Boltzmann machine itself is an undirected graphical model based on a bipartite graph, with visible units in one part of the graph and hidden units in the other part. There are no connections among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

We begin with the binary version of the restricted Boltzmann machine, but as we see later there are extensions to other types of visible and hidden units.

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

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

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

The energy function for an RBM is given by

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

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

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

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

20.2.1 Conditional Distributions

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

Deriving the conditional distributions from the joint distribution is straightforward:

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

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

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

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

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

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

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

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

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

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

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

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

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

20.2.2 Training Restricted Boltzmann Machines

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

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

20.3 Deep Belief Networks

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

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

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

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

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

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

In the case of real-valued visible units, substitute

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

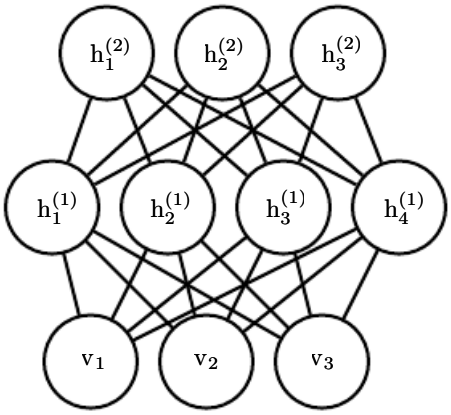


Figure 20.2: The graphical model for a deep Boltzmann machine with one visible layer (bottom) and two hidden layers. Connections are only between units in neighboring layers. There are no intralayer layer connections.

20.4 Deep Boltzmann Machines

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

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

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

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

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

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

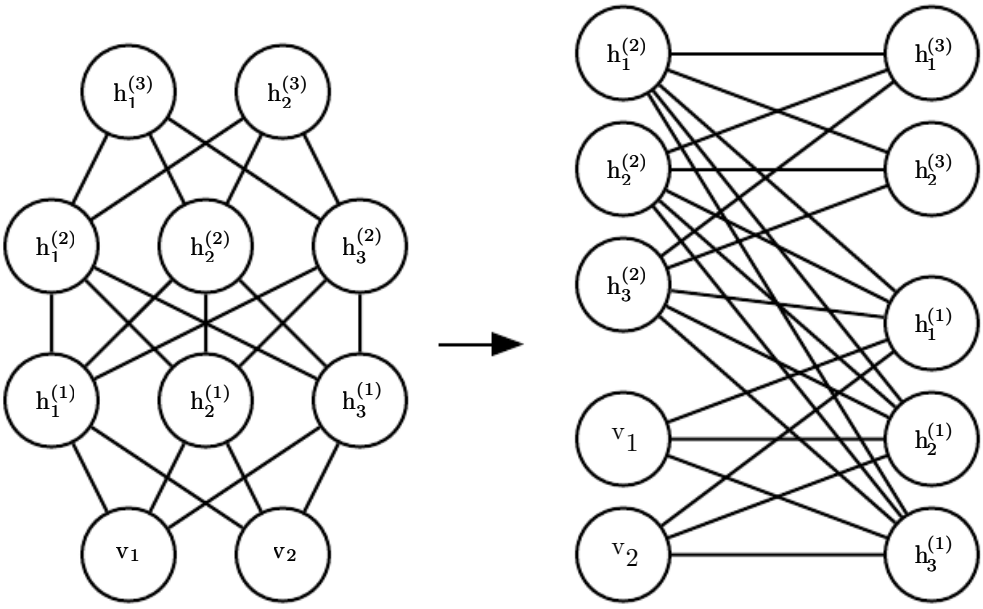


Figure 20.3: A deep Boltzmann machine, re-arranged to reveal its bipartite graph structure.

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

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

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

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

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

and

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

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

20.4.1 Interesting Properties

Deep Boltzmann machines have many interesting properties.

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

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

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

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

20.4.2 DBM Mean Field Inference

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

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

In variational approximations to inference, we approach the task of approxi-

mating a particular target distribution—in our case, the posterior distribution over the hidden units given the visible units—by some reasonably simple family of distributions. In the case of the mean field approximation, the approximating family is the set of distributions where the hidden units are conditionally independent.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

20.4.3 DBM Parameter Learning

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

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

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

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

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

The non-variational version of stochastic maximum likelihood algorithm was discussed earlier, in section 18.2. Variational stochastic maximum likelihood as applied to the DBM is given in algorithm 20.1. Recall that we describe a simplified varient of the DBM that lacks bias parameters; including them is trivial.

20.4.4 Layer-Wise Pretraining

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

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

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

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

RBMs have been trained in this way, they can be combined to form a DBM. The DBM may then be trained with PCD. Typically PCD training will make only a small change in the model's parameters and its performance as measured by the log-likelihood it assigns to the data, or its ability to classify inputs. See figure 20.4 for an illustration of the training procedure.

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

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

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

20.4.5 Jointly Training Deep Boltzmann Machines

Classic DBMs require greedy unsupervised pretraining, and to perform classification well, require a separate MLP-based classifier on top of the hidden features they extract. This has some undesirable properties. It is hard to track performance during training because we cannot evaluate properties of the full DBM while training the first RBM. Thus, it is hard to tell how well our hyperparameters

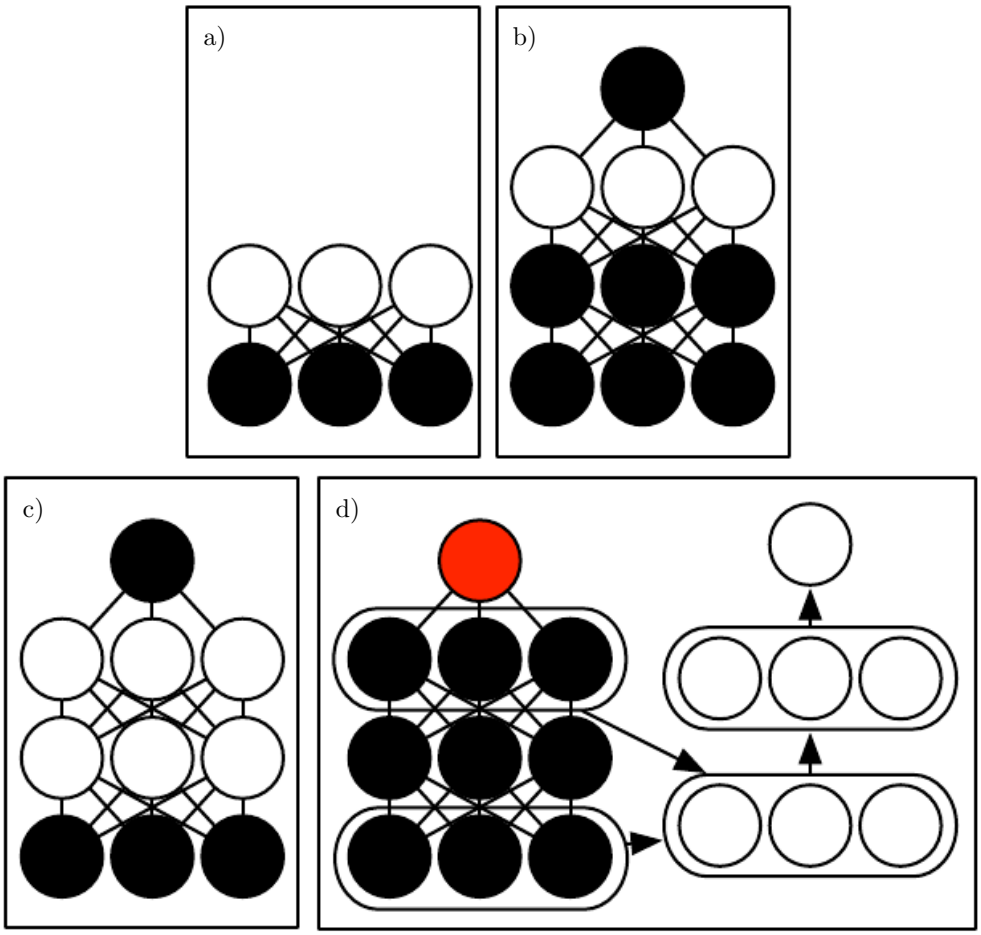


Figure 20.4: The deep Boltzmann machine training procedure used to classify the MNIST dataset (Salakhutdinov and Hinton, 2009a; Srivastava *et al.*, 2014). *(a)*Train an RBM by using CD to approximately maximize $\log P(\boldsymbol{v})$. *(b)*Train a second RBM that models $\boldsymbol{h}^{(1)}$ and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

are working until quite late in the training process. Software implementations of DBMs need to have many different components for CD training of individual RBMs, PCD training of the full DBM, and training based on back-propagation through the MLP. Finally, the MLP on top of the Boltzmann machine loses many of the advantages of the Boltzmann machine probabilistic model, such as being able to perform inference when some input values are missing.

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

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

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

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

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

Typically $\boldsymbol{\mu}$ is a hyperparameter fixed at the beginning of training. It is usually chosen to make sure that $\boldsymbol{x} - \boldsymbol{\mu} \approx \boldsymbol{0}$ when the model is initialized. This reparametrization does not change the set of probability distributions that the model can represent, but it does change the dynamics of stochastic gradient descent applied to the likelihood. Specifically, in many cases, this reparametrization results

in a Hessian matrix that is better conditioned. Melchior *et al.* (2013) experimentally confirmed that the conditioning of the Hessian matrix improves, and observed that the centering trick is equivalent to another Boltzmann machine learning technique, the **enhanced gradient** (Cho *et al.*, 2011). The improved conditioning of the Hessian matrix allows learning to succeed, even in difficult cases like training a deep Boltzmann machine with multiple layers.

The other approach to jointly training deep Boltzmann machines is the multi-prediction deep Boltzmann machine (MP-DBM) which works by viewing the mean field equations as defining a family of recurrent networks for approximately solving every possible inference problem (Goodfellow *et al.*, 2013b). Rather than training the model to maximize the likelihood, the model is trained to make each recurrent network obtain an accurate answer to the corresponding inference problem. The training process is illustrated in figure 20.5. It consists of randomly sampling a training example, randomly sampling a subset of inputs to the inference network, and then training the inference network to predict the values of the remaining units.

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

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

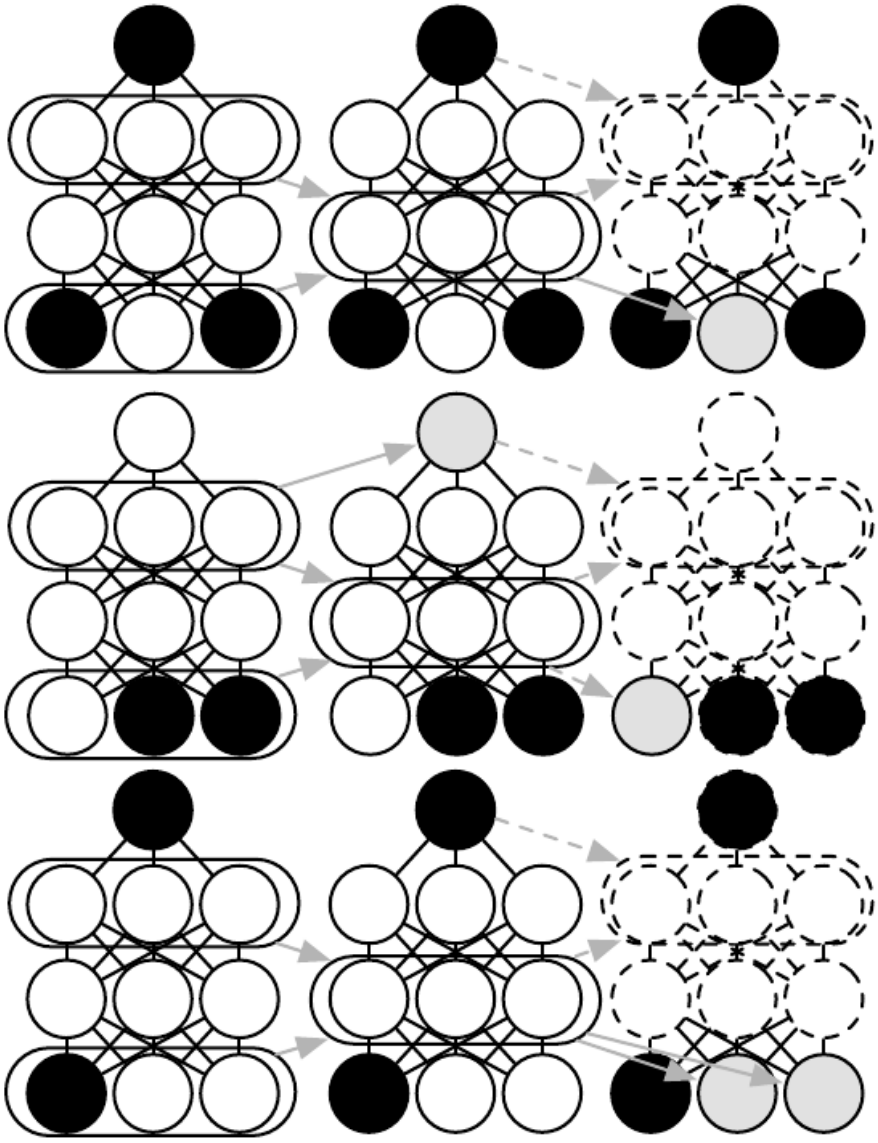


Figure 20.5: An illustration of the multi-prediction training process for a deep Boltzmann machine. Each row indicates a different example within a minibatch for the same training step. Each column represents a time step within the mean field inference process. For each example, we sample a subset of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

DBMs may be trained jointly while DBMs require a greedy layer-wise pretraining. The disadvantage of back-propagating through the approximate inference graph is that it does not provide a way to optimize the log-likelihood, but rather a heuristic approximation of the generalized pseudolikelihood.

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

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

20.5 Boltzmann Machines for Real-Valued Data

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

20.5.1 Gaussian-Bernoulli RBMs

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

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

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

We can find the terms we need to add to the energy function by expanding the unnormalized log conditional distribution:

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

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

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

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

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

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

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

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

we include bias parameters for the hidden units) but it does affect the learning dynamics of the model. Including the term may help the hidden unit activations remain reasonable even when the weights rapidly increase in magnitude.

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

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

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

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

20.5.2 Undirected Models of Conditional Covariance

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

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

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

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

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

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

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

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

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

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

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

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

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

Note that the covariance matrix $\boldsymbol{C}^{\text{mc}}_{\boldsymbol{x}|\boldsymbol{h}} = \left(\sum_j h_j^{(c)} \boldsymbol{r}^{(j)} \boldsymbol{r}^{(j)\top} + \boldsymbol{I}\right)^{-1}$ is non-diagonal and that $\boldsymbol{W}$ is the weight matrix associated with the Gaussian RBM modeling the

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

conditional means. It is difficult to train the mcRBM via contrastive divergence or persistent contrastive divergence because of its non-diagonal conditional covariance structure. CD and PCD require sampling from the joint distribution of $\boldsymbol{x}, \boldsymbol{h}^{(m)}, \boldsymbol{h}^{(c)}$ which, in a standard RBM, is accomplished by Gibbs sampling over the conditionals. However, in the mcRBM, sampling from $p_{\text{mc}}(\boldsymbol{x} \mid \boldsymbol{h}^{(m)}, \boldsymbol{h}^{(c)})$ requires computing $(\boldsymbol{C}^{\text{mc}})^{-1}$ at every iteration of learning. This can be an impractical computational burden for larger observations. Ranzato and Hinton (2010) avoid direct sampling from the conditional $p_{\text{mc}}(\boldsymbol{x} \mid \boldsymbol{h}^{(m)}, \boldsymbol{h}^{(c)})$ by sampling directly from the marginal $p(\boldsymbol{x})$ using Hamiltonian (hybrid) Monte Carlo (Neal, 1993) on the mcRBM free energy.

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

The mPoT energy function is:

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

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

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

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

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

The spike and slab RBM has two sets of hidden units: binary **spike** units $\mathbf{h}$, and real-valued **slab** units $\mathbf{s}$. The mean of the visible units conditioned on the hidden units is given by $(\boldsymbol{h} \odot \boldsymbol{s})\boldsymbol{W}^\top$. In other words, each column $\boldsymbol{W}_{:,i}$ defines a component that can appear in the input when $h_i = 1$. The corresponding spike variable h_i determines whether that component is present at all. The corresponding slab variable s_i determines the intensity of that component, if it is present. When a spike variable is active, the corresponding slab variable adds variance to the input along the axis defined by $\boldsymbol{W}_{:,i}$. This allows us to model the covariance of the inputs. Fortunately, contrastive divergence and persistent contrastive divergence with Gibbs sampling are still applicable. There is no need to invert any matrix.

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

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

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

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

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

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

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

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

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

Comparing the ssRBM to the mcRBM and the mPoT models, the ssRBM parametrizes the conditional covariance of the observation in a significantly different way. The mcRBM and mPoT both model the covariance structure of the observation as $\left(\sum_j h_j^{(c)} \boldsymbol{r}^{(j)} \boldsymbol{r}^{(j)\top} + \boldsymbol{I}\right)^{-1}$, using the activation of the hidden units $\boldsymbol{h}_j > 0$ to enforce constraints on the conditional covariance in the direction $\boldsymbol{r}^{(j)}$. In contrast, the ssRBM specifies the conditional covariance of the observations using the hidden spike activations $h_i = 1$ to pinch the precision matrix along the direction specified by the corresponding weight vector. The ssRBM conditional covariance is very similar to that given by a different model: the product of probabilistic principal components analysis (PoPPCA) (Williams and Agakov, 2002). In the overcomplete setting, sparse activations with the ssRBM parametrization permit significant variance (above the nominal variance given by $\boldsymbol{\Lambda}^{-1}$) only in the selected directions of the sparsely activated h_i. In the mcRBM or mPoT models, an overcomplete representation would mean that to capture variation in a particular direction in the observation space requires removing potentially all constraints with positive projection in that direction. This would suggest that these models are less well suited to the overcomplete setting.

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

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

The ssRBM allows for several extensions. Including higher-order interactions and average-pooling of the slab variables (Courville *et al.*, 2014) enables the model to learn excellent features for a classifier when labeled data is scarce. Adding a

term to the energy function that prevents the partition function from becoming undefined results in a sparse coding model, spike and slab sparse coding (Goodfellow *et al.*, 2013d), also known as S3C.

20.6 Convolutional Boltzmann Machines

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

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

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

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

machines.

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

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

Pixels at the boundary of the image also pose some difficulty, which is exacerbated by the fact that connections in a Boltzmann machine are symmetric. If we do not implicitly zero-pad the input, then there are fewer hidden units than visible units, and the visible units at the boundary of the image are not modeled

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

well because they lie in the receptive field of fewer hidden units. However, if we do implicitly zero-pad the input, then the hidden units at the boundary are driven by fewer input pixels, and may fail to activate when needed.

20.7 Boltzmann Machines for Structured or Sequential Outputs

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

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

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

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

Another sequence modeling task is to model the distribution over sequences

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

20.8 Other Boltzmann Machines

Many other variants of Boltzmann machines are possible.

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

Most Boltzmann machines used in practice have only second-order interactions in their energy functions, meaning that their energy functions are the sum of many terms and each individual term only includes the product between two random variables. An example of such a term is $v_i W_{i,j} h_j$. It is also possible to train higher-order Boltzmann machines (Sejnowski, 1987) whose energy function terms involve the products between many variables. Three-way interactions between a hidden unit and two different images can model spatial transformations from one frame of video to the next (Memisevic and Hinton, 2007, 2010). Multiplication by a one-hot class variable can change the relationship between visible and hidden units depending on which class is present (Nair and Hinton, 2009). One recent example of the use of higher-order interactions is a Boltzmann machine with two groups of hidden units, with one group of hidden units that interact with both the visible units $\boldsymbol{v}$ and the class label y, and another group of hidden units that interact only with the $\boldsymbol{v}$ input values (Luo *et al.*, 2011). This can be interpreted as encouraging

some hidden units to learn to model the input using features that are relevant to the class but also to learn extra hidden units that explain nuisance details that are necessary for the samples of $\boldsymbol{v}$ to be realistic but do not determine the class of the example. Another use of higher-order interactions is to gate some features. Sohn *et al.* (2013) introduced a Boltzmann machine with third-order interactions with binary mask variables associated with each visible unit. When these masking variables are set to zero, they remove the influence of a visible unit on the hidden units. This allows visible units that are not relevant to the classification problem to be removed from the inference pathway that estimates the class.

More generally, the Boltzmann machine framework is a rich space of models permitting many more model structures than have been explored so far. Developing a new form of Boltzmann machine requires some more care and creativity than developing a new neural network layer, because it is often difficult to find an energy function that maintains tractability of all of the different conditional distributions needed to use the Boltzmann machine, but despite this required effort the field remains open to innovation.

20.9 Back-Propagation through Random Operations

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

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

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

Because an individual sample of y is not produced by a function, but rather by a sampling process whose output changes every time we query it, it may seem counterintuitive to take the derivatives of y with respect to the parameters of its distribution, μ and σ^2. However, we can rewrite the sampling process as

transforming an underlying random value $\mathrm{z} \sim \mathcal{N}(z; 0, 1)$ to obtain a sample from the desired distribution:

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

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

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

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

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

as

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

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

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

In neural network applications, we typically choose $\boldsymbol{z}$ to be drawn from some simple distribution, such as a unit uniform or unit Gaussian distribution, and achieve more complex distributions by allowing the deterministic portion of the network to reshape its input.

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

20.9.1 Back-Propagating through Discrete Stochastic Operations

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

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

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

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

The simplest version of REINFORCE can be derived by simply differentiating the expected cost:

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

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

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

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

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

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

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

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

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

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

which means that

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

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

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

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

The gradient estimator with respect to ω_i then becomes

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

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

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

magnitude of this quantity. Mnih and Gregor (2014) called this heuristic **variance normalization**.

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

20.10 Directed Generative Nets

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

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

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

20.10.1 Sigmoid Belief Nets

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

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

The most common structure of sigmoid belief network is one that is divided into many layers, with ancestral sampling proceeding through a series of many hidden layers and then ultimately generating the visible layer. This structure is very similar to the deep belief network, except that the units at the beginning of

the sampling process are independent from each other, rather than sampled from a restricted Boltzmann machine. Such a structure is interesting for a variety of reasons. One reason is that the structure is a universal approximator of probability distributions over the visible units, in the sense that it can approximate any probability distribution over binary variables arbitrarily well, given enough depth, even if the width of the individual layers is restricted to the dimensionality of the visible layer (Sutskever and Hinton, 2008).

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

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

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

20.10.2 Differentiable Generator Nets

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Generative modeling seems to be more difficult than classification or regression because the learning process requires optimizing intractable criteria. In the context

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

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

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

20.10.3 Variational Autoencoders

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

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

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

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

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

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

In equation 20.76, we recognize the first term as the joint log-likelihood of the visible and hidden variables under the approximate posterior over the latent variables (just like with EM, except that we use an approximate rather than the exact posterior). We recognize also a second term, the entropy of the approximate posterior. When q is chosen to be a Gaussian distribution, with noise added to a predicted mean value, maximizing this entropy term encourages increasing the standard deviation of this noise. More generally, this entropy term encourages the variational posterior to place high probability mass on many $\boldsymbol{z}$ values that could have generated $\boldsymbol{x}$, rather than collapsing to a single point estimate of the most likely value. In equation 20.77, we recognize the first term as the reconstruction log-likelihood found in other autoencoders. The second term tries to make the approximate posterior distribution $q(\mathbf{z} \mid \boldsymbol{x})$ and the model prior $p_{\text{model}}(\boldsymbol{z})$ approach each other.

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

The variational autoencoder approach is elegant, theoretically pleasing, and simple to implement. It also obtains excellent results and is among the state of the art approaches to generative modeling. Its main drawback is that samples from variational autoencoders trained on images tend to be somewhat blurry. The causes of this phenomenon are not yet known. One possibility is that the blurriness is an intrinsic effect of maximum likelihood, which minimizes $D_{\mathrm{KL}}(p_{\text{data}} \| p_{\text{model}})$. As illustrated in figure 3.6, this means that the model will assign high probability to points that occur in the training set, but may also assign high probability to other points. These other points may include blurry images. Part of the reason that the model would choose to put probability mass on blurry images rather than some other part of the space is that the variational autoencoders used in practice usually have a Gaussian distribution for $p_{\text{model}}(\boldsymbol{x}; g(\boldsymbol{z}))$. Maximizing a lower bound on the likelihood of such a distribution is similar to training a traditional autoencoder with mean squared error, in the sense that it has a tendency to ignore features of the input that occupy few pixels or that cause only a small change in the brightness of the pixels that they occupy. This issue is not specific to VAEs and

is shared with generative models that optimize a log-likelihood, or equivalently, $D_{\mathrm{KL}}(p_{\mathrm{data}} \| p_{\mathrm{model}})$, as argued by Theis *et al.* (2015) and by Huszar (2015). Another troubling issue with contemporary VAE models is that they tend to use only a small subset of the dimensions of $\boldsymbol{z}$, as if the encoder was not able to transform enough of the local directions in input space to a space where the marginal distribution matches the factorized prior.

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

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

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

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

Variational autoencoders have some interesting connections to the MP-DBM and other approaches that involve back-propagation through the approximate inference graph (Goodfellow *et al.*, 2013b; Stoyanov *et al.*, 2011; Brakel *et al.*, 2013). These previous approaches required an inference procedure such as mean field fixed point equations to provide the computational graph. The variational autoencoder is defined for arbitrary computational graphs, which makes it applicable to a wider range of probabilistic model families because there is no need to restrict the choice

of models to those with tractable mean field fixed point equations. The variational autoencoder also has the advantage that it increases a bound on the log-likelihood of the model, while the criteria for the MP-DBM and related models are more heuristic and have little probabilistic interpretation beyond making the results of approximate inference accurate. One disadvantage of the variational autoencoder is that it learns an inference network for only one problem, inferring $\boldsymbol{z}$ given $\boldsymbol{x}$. The older methods are able to perform approximate inference over any subset of variables given any other subset of variables, because the mean field fixed point equations specify how to share parameters between the computational graphs for all of these different problems.

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

20.10.4 Generative Adversarial Networks

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

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

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

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

The default choice for v is

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

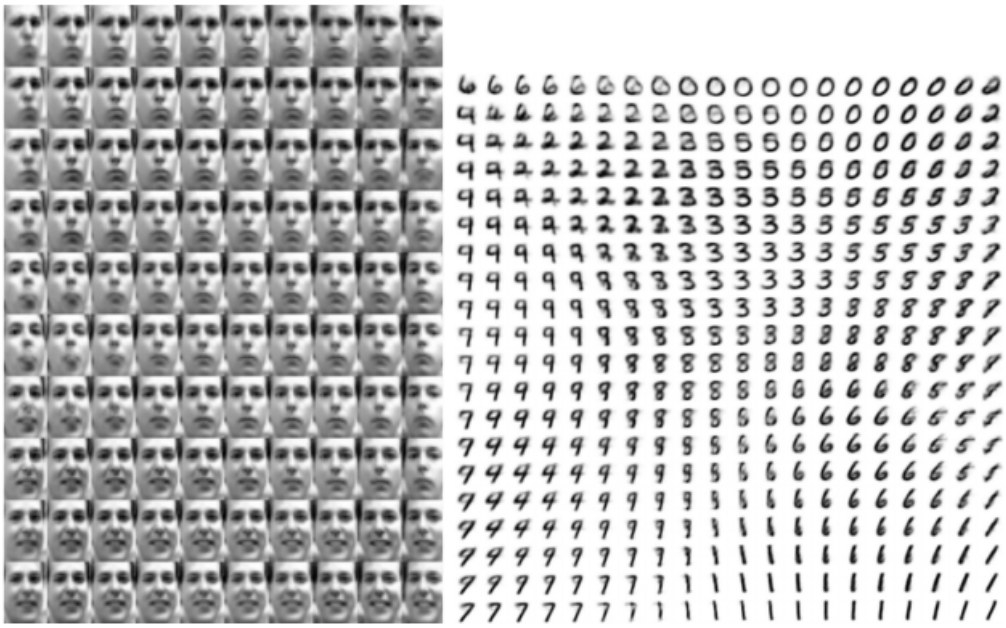


Figure 20.6: Examples of two-dimensional coordinate systems for high-dimensional manifolds, learned by a variational autoencoder (Kingma and Welling, 2014a). Two dimensions may be plotted directly on the page for visualization, so we can gain an understanding of how the model works by training a model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

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

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

Unfortunately, learning in GANs can be difficult in practice when g and d are represented by neural networks and $\max_d v(g, d)$ is not convex. Goodfellow

(2014) identified non-convergence as an issue that may cause GANs to underfit. In general, simultaneous gradient descent on two players' costs is not guaranteed to reach an equilibrium. Consider for example the value function $v(a, b) = ab$, where one player controls a and incurs cost ab, while the other player controls b and receives a cost $-ab$. If we model each player as making infinitesimally small gradient steps, each player reducing their own cost at the expense of the other player, then a and b go into a stable, circular orbit, rather than arriving at the equilibrium point at the origin. Note that the equilibria for a minimax game are not local minima of v. Instead, they are points that are simultaneously minima for both players' costs. This means that they are saddle points of v that are local minima with respect to the first player's parameters and local maxima with respect to the second player's parameters. It is possible for the two players to take turns increasing then decreasing v forever, rather than landing exactly on the saddle point where neither player is capable of reducing its cost. It is not known to what extent this non-convergence problem affects GANs.

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

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

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

The GAN learning problem can also be simplified by breaking the generation

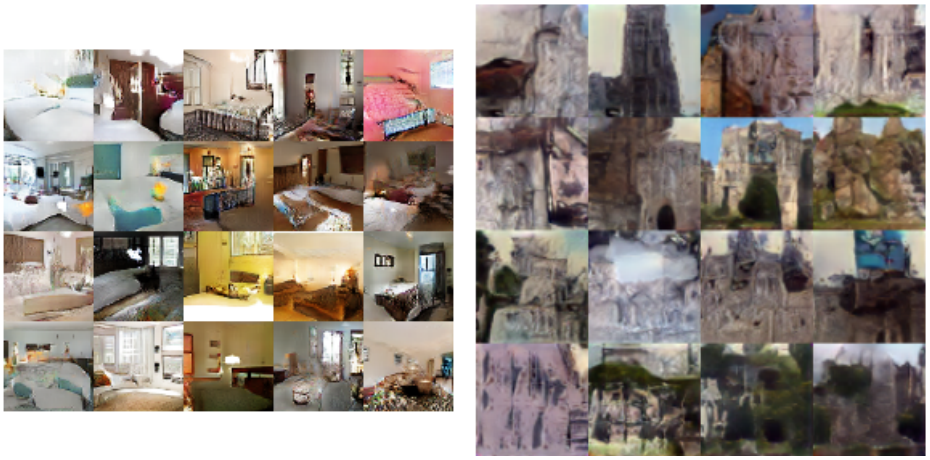


Figure 20.7: Images generated by GANs trained on the LSUN dataset. *(Left)*Images of bedrooms generated by a DCGAN model, reproduced with permission from Radford *et al.* (2015). *(Right)*Images of churches generated by a LAPGAN model, reproduced with permission from Denton *et al.* (2015).

process into many levels of detail. It is possible to train conditional GANs (Mirza and Osindero, 2014) that learn to sample from a distribution $p(\boldsymbol{x} \mid \boldsymbol{y})$ rather than simply sampling from a marginal distribution $p(\boldsymbol{x})$. Denton *et al.* (2015) showed that a series of conditional GANs can be trained to first generate a very low-resolution version of an image, then incrementally add details to the image. This technique is called the LAPGAN model, due to the use of a Laplacian pyramid to generate the images containing varying levels of detail. LAPGAN generators are able to fool not only discriminator networks but also human observers, with experimental subjects identifying up to 40% of the outputs of the network as being real data. See figure 20.7 for examples of images generated by a LAPGAN generator.

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

Gaussian distribution.

Dropout seems to be important in the discriminator network. In particular, units should be stochastically dropped while computing the gradient for the generator network to follow. Following the gradient of the deterministic version of the discriminator with its weights divided by two does not seem to be as effective. Likewise, never using dropout seems to yield poor results.

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

20.10.5 Generative Moment Matching Networks

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

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

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

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

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

dynamically updated discriminator that automatically focuses its attention on whichever statistic the generator network is matching the least effectively.

Instead, generative moment matching networks can be trained by minimizing a cost function called **maximum mean discrepancy** (Schölkopf and Smola, 2002; Gretton *et al.*, 2012) or MMD. This cost function measures the error in the first moments in an infinite-dimensional space, using an implicit mapping to feature space defined by a kernel function in order to make computations on infinite-dimensional vectors tractable. The MMD cost is zero if and only if the two distributions being compared are equal.

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

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

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

20.10.6 Convolutional Generative Networks

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

Convolutional networks for recognition tasks have information flow from the image to some summarization layer at the top of the network, often a class label.

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

20.10.7 Auto-Regressive Networks

Auto-regressive networks are directed probabilistic models with no latent random variables. The conditional probability distributions in these models are represented by neural networks (sometimes extremely simple neural networks such as logistic regression). The graph structure of these models is the complete graph. They decompose a joint probability over the observed variables using the chain rule of probability to obtain a product of conditionals of the form $P(x_d \mid x_{d-1}, \ldots, x_1)$. Such models have been called **fully-visible Bayes networks** (FVBNs) and used successfully in many forms, first with logistic regression for each conditional distribution (Frey, 1998) and then with neural networks with hidden units (Bengio and Bengio, 2000b; Larochelle and Murray, 2011). In some forms of auto-regressive networks, such as NADE (Larochelle and Murray, 2011), described

in section 20.10.10 below, we can introduce a form of parameter sharing that brings both a statistical advantage (fewer unique parameters) and a computational advantage (less computation). This is one more instance of the recurring deep learning motif of *reuse of features*.

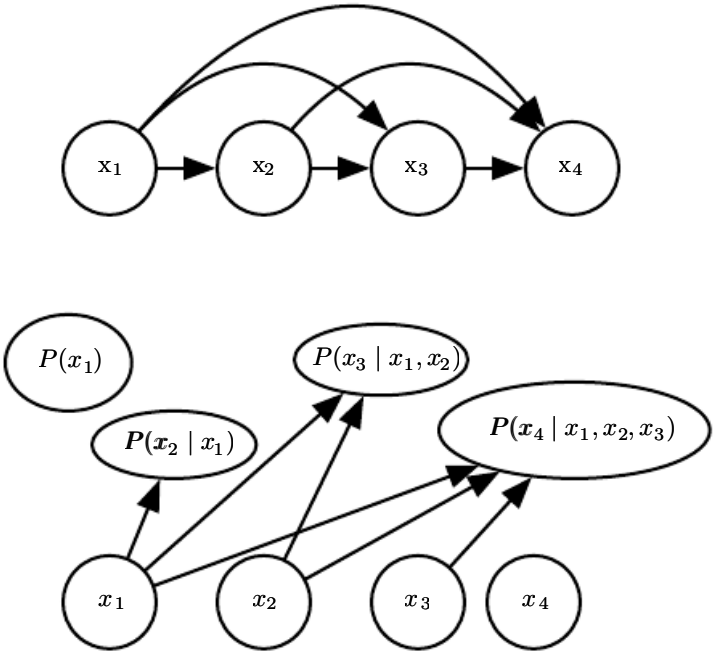


Figure 20.8: A fully visible belief network predicts the i-th variable from the $i-1$ previous ones. *(Top)*The directed graphical model for an FVBN. *(Bottom)*Corresponding computational graph, in the case of the logistic FVBN, where each prediction is made by a linear predictor.

20.10.8 Linear Auto-Regressive Networks

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

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

Linear auto-regressive networks are essentially the generalization of linear classification methods to generative modeling. They therefore have the same

advantages and disadvantages as linear classifiers. Like linear classifiers, they may be trained with convex loss functions, and sometimes admit closed form solutions (as in the Gaussian case). Like linear classifiers, the model itself does not offer a way of increasing its capacity, so capacity must be raised using techniques like basis expansions of the input or the kernel trick.

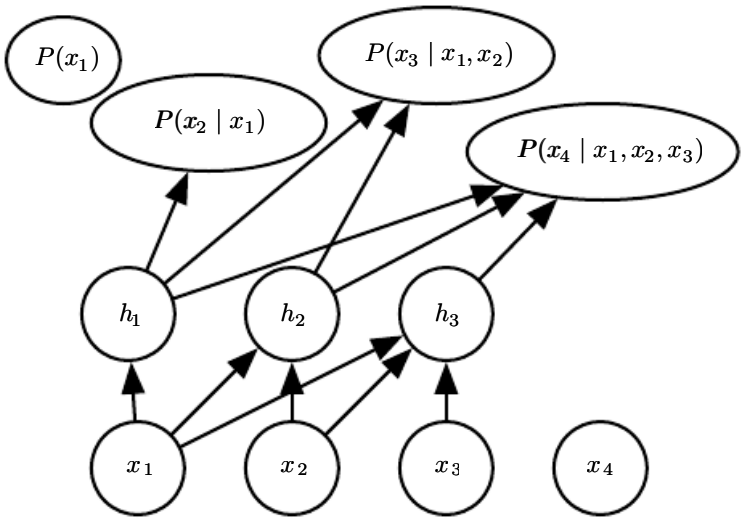


Figure 20.9: A neural auto-regressive network predicts the i-th variable x_i from the $i-1$ previous ones, but is parametrized so that features (groups of hidden units denoted h_i) that are functions of $x_1, \ldots, x_i$ can be reused in predicting all of the subsequent variables $x_{i+1}, x_{i+2},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

20.10.9 Neural Auto-Regressive Networks

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

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

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

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

20.10.10 NADE

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

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

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

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

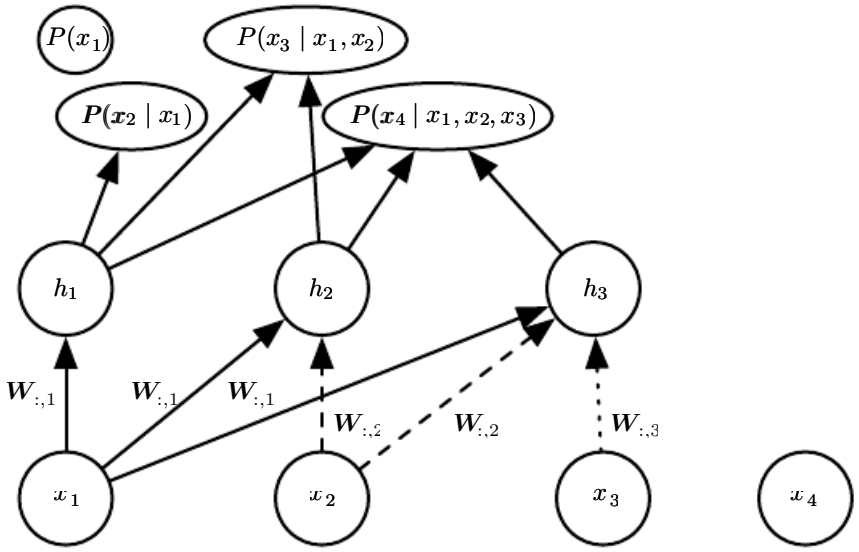


Figure 20.10: An illustration of the neural autoregressive density estimator (NADE). The hidden units are organized in groups $\boldsymbol{h}^{(j)}$ so that only the inputs $x_1, \ldots, x_i$ participate in computing $\boldsymbol{h}^{(i)}$ and predicting $P(x_j \mid x_{j-1}, \ldots, x_1)$, for $j > i$.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

Larochelle and Murray (2011) chose this sharing scheme so that forward propagation in a NADE model loosely resembles the computations performed in mean field inference to fill in missing inputs in an RBM. This mean field inference corresponds to running a recurrent network with shared weights and the first step of that inference is the same as in NADE. The only difference is that with NADE, the output weights connecting the hidden units to the output are parametrized independently from the weights connecting the input units to the hidden units. In the RBM, the hidden-to-output weights are the transpose of the input-to-hidden weights. The NADE architecture can be extended to mimic not just one time step of the mean field recurrent inference but to mimic k steps. This approach is called NADE-k (Raiko *et al.*, 2014).

As mentioned previously, auto-regressive networks may be extend to process continuous-valued data. A particularly powerful and generic way of parametrizing a continuous density is as a Gaussian mixture (introduced in section 3.9.6) with mixture weights α_i (the coefficient or prior probability for component i), per-component conditional mean μ_i and per-component conditional variance σ_i^2. A model called RNADE (Uria *et al.*, 2013) uses this parametrization to extend NADE to real values. As with other mixture density networks, the parameters of this

distribution are outputs of the network, with the mixture weight probabilities produced by a softmax unit, and the variances parametrized so that they are positive. Stochastic gradient descent can be numerically ill-behaved due to the interactions between the conditional means μ_i and the conditional variances σ_i^2. To reduce this difficulty, Uria *et al.* (2013) use a pseudo-gradient that replaces the gradient on the mean, in the back-propagation phase.

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

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

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

In the same paper, the authors propose deep versions of the architecture, but unfortunately that immediately makes computation as expensive as in the original neural auto-regressive neural network (Bengio and Bengio, 2000b). The first layer and the output layer can still be computed in $O(nh)$ multiply-add operations, as in the regular NADE, where h is the number of hidden units (the size of the groups h_i, in figures 20.10 and 20.9), 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 Drawing Samples from Autoencoders

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

Some kinds of autoencoders, such as the variational autoencoder, explicitly represent a probability distribution and admit straightforward ancestral sampling. Most other kinds of autoencoders require MCMC sampling.

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

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

20.11.1 Markov Chain Associated with any Denoising Autoencoder

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

Each step of the Markov chain that generates from the estimated distribution consists of the following sub-steps, illustrated in figure 20.11:

1. Starting from the previous state $\boldsymbol{x}$, inject corruption noise, sampling $\tilde{\boldsymbol{x}}$ from $C(\tilde{\boldsymbol{x}} \mid \boldsymbol{x})$.
2. Encode $\tilde{\boldsymbol{x}}$ into $\boldsymbol{h} = f(\tilde{\boldsymbol{x}})$.
3. Decode $\boldsymbol{h}$ to obtain the parameters $\boldsymbol{\omega} = g(\boldsymbol{h})$ of $p(\mathbf{x} \mid \boldsymbol{\omega} = g(\boldsymbol{h})) = p(\mathbf{x} \mid \tilde{\boldsymbol{x}})$.
4. Sample the next state $\boldsymbol{x}$ from $p(\mathbf{x} \mid \boldsymbol{\omega} = g(\boldsymbol{h})) = p(\mathbf{x} \mid \tilde{\boldsymbol{x}})$.

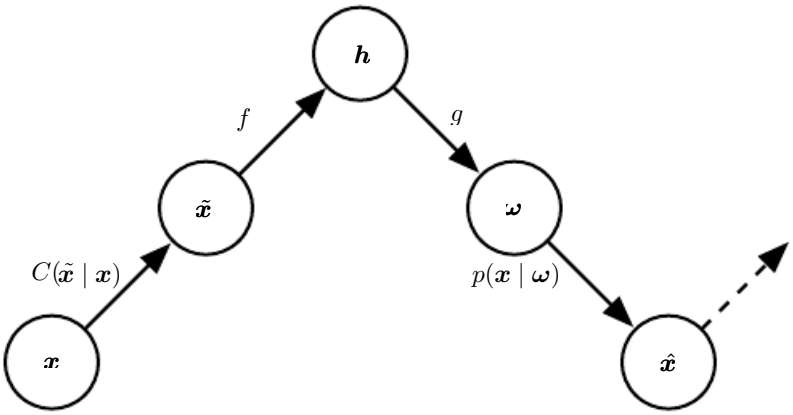


Figure 20.11: Each step of the Markov chain associated with a trained denoising autoencoder, that generates the samples from the probabilistic model implicitly trained by the denoising log-likelihood criterion. Each step consists in (a) injecting noise via corruption process C in state $\boldsymbol{x}$, yielding $\tilde{\boldsymbol{x}}$, (b) encoding it with function f, yielding $\boldsymbol{h} = f(\tilde{\boldsymbol{x}})$, (c) decoding the result with function g, yielding parameters $\boldsymbol{\omega}$ for the reconstruction distribution, and (d) given $\boldsymbol{\omega}$, sampling a new state from the reconstruction distribution $p(\mathbf{x} \mid \boldsymbol{\omega} = g(f(\tilde{\boldsymbol{x}})))$. In the typical squared reconstruction error case, $g(\boldsymbol{h}) = \hat{\boldsymbol{x}}$, which estimates $\mathbb{E}[\boldsymbol{x} \mid \tilde{\boldsymbol{x}}]$, corruption consists in adding Gaussian noise and sampling from $p(\mathbf{x} \mid \boldsymbol{\omega})$ consists in adding Gaussian noise, a second time, to the reconstruction $\hat{\boldsymbol{x}}$. The latter noise level should correspond to the mean squared error of reconstructions, whereas the injected noise is a hyperparameter that controls the mixing speed as well as the extent to which the estimator smooths the empirical distribution (Vincent, 2011). In the example illustrated here, only the C and p conditionals are stochastic steps (f and g are deterministic computations), although noise can also be injected inside the autoencoder, as in generative stochastic networks (Bengio *et al.*, 2014).

Bengio *et al.* (2014) showed that if the autoencoder $p(\mathbf{x} \mid \tilde{\mathbf{x}})$ forms a consistent estimator of the corresponding true conditional distribution, then the stationary distribution of the above Markov chain forms a consistent estimator (albeit an implicit one) of the data generating distribution of $\mathbf{x}$.

20.11.2 Clamping and Conditional Sampling

Similarly to Boltzmann machines, denoising autoencoders and their generalizations (such as GSNs, described below) can be used to sample from a conditional distribution $p(\mathbf{x}_f \mid \mathbf{x}_o)$, simply by clamping the *observed* units $\mathbf{x}_f$ and only resampling the *free* units $\mathbf{x}_o$ given $\mathbf{x}_f$ and the sampled latent variables (if any). For example, MP-DBMs can be interpreted as a form of denoising autoencoder, and are able to sample missing inputs. GSNs later generalized some of the ideas present in MP-DBMs to perform the same operation (Bengio *et al.*, 2014). Alain *et al.* (2015) identified a missing condition from Proposition 1 of Bengio *et al.* (2014), which is that the transition operator (defined by the stochastic mapping going from one state of the chain to the next) should satisfy a property called **detailed balance**, which specifies that a Markov Chain at equilibrium will remain in equilibrium whether the transition operator is run in forward or reverse.

An experiment in clamping half of the pixels (the right part of the image) and running the Markov chain on the other half is shown in figure 20.12.

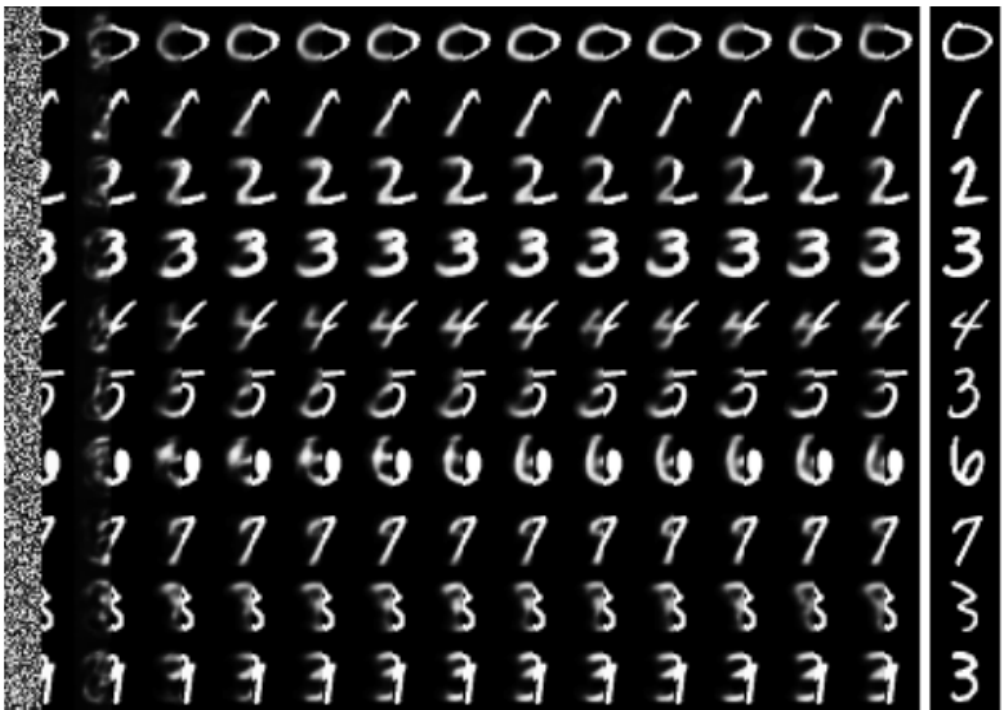


Figure 20.12: Illustration of clamping the right half of the image and running the Markov Chain by resampling only the left half at each step. These samples come from a GSN trained to reconstruct MNIST digits at each time step using the walkback procedure.

20.11.3 Walk-Back Training Procedure

The walk-back training procedure was proposed by Bengio *et al.* (2013c) as a way to accelerate the convergence of generative training of denoising autoencoders. Instead of performing a one-step encode-decode reconstruction, this procedure consists in alternative multiple stochastic encode-decode steps (as in the generative Markov chain) initialized at a training example (just like with the contrastive divergence algorithm, described in section 18.2) and penalizing the last probabilistic reconstructions (or all of the reconstructions along the way).

Training with k steps is equivalent (in the sense of achieving the same stationary distribution) as training with one step, but practically has the advantage that spurious modes further from the data can be removed more efficiently.

20.12 Generative Stochastic Networks

Generative stochastic networks or GSNs (Bengio *et al.*, 2014) are generalizations of denoising autoencoders that include latent variables $\mathbf{h}$ in the generative

Markov chain, in addition to the visible variables (usually denoted $\mathbf{x}$).

A GSN is parametrized by two conditional probability distributions which specify one step of the Markov chain:

1. $p(\mathbf{x}^{(k)} \mid \mathbf{h}^{(k)})$ tells how to generate the next visible variable given the current latent state. Such a "reconstruction distribution" is also found in denoising autoencoders, RBMs, DBNs and DBMs.
2. $p(\mathbf{h}^{(k)} \mid \mathbf{h}^{(k-1)}, \mathbf{x}^{(k-1)})$ tells how to update the latent state variable, given the previous latent state and visible variable.

Denoising autoencoders and GSNs differ from classical probabilistic models (directed or undirected) in that they parametrize the generative process itself rather than the mathematical specification of the joint distribution of visible and latent variables. Instead, the latter is defined *implicitly*, *if it exists*, as the stationary distribution of the generative Markov chain. The conditions for existence of the stationary distribution are mild and are the same conditions required by standard MCMC methods (see section 17.3). These conditions are necessary to guarantee that the chain mixes, but they can be violated by some choices of the transition distributions (for example, if they were deterministic).

One could imagine different training criteria for GSNs. The one proposed and evaluated by Bengio *et al.* (2014) is simply reconstruction log-probability on the visible units, just like for denoising autoencoders. This is achieved by clamping $\mathbf{x}^{(0)} = \boldsymbol{x}$ to the observed example and maximizing the probability of generating $\boldsymbol{x}$ at some subsequent time steps, i.e., maximizing $\log p(\mathbf{x}^{(k)} = \boldsymbol{x} \mid \mathbf{h}^{(k)})$, where $\mathbf{h}^{(k)}$ is sampled from the chain, given $\mathbf{x}^{(0)} = \boldsymbol{x}$. In order to estimate the gradient of $\log p(\mathbf{x}^{(k)} = \boldsymbol{x} \mid \mathbf{h}^{(k)})$ with respect to the other pieces of the model, Bengio *et al.* (2014) use the reparametrization trick, introduced in section 20.9.

The **walk-back training** protocol (described in section 20.11.3) was used (Bengio *et al.*, 2014) to improve training convergence of GSNs.

20.12.1 Discriminant GSNs

The original formulation of GSNs (Bengio *et al.*, 2014) was meant for unsupervised learning and implicitly modeling $p(\mathbf{x})$ for observed data $\mathbf{x}$, but it is possible to modify the framework to optimize $p(\mathbf{y} \mid \boldsymbol{x})$.

For example, Zhou and Troyanskaya (2014) generalize GSNs in this way, by only back-propagating the reconstruction log-probability over the output variables, keeping the input variables fixed. They applied this successfully to model sequences

(protein secondary structure) and introduced a (one-dimensional) convolutional structure in the transition operator of the Markov chain. It is important to remember that, for each step of the Markov chain, one generates a new sequence for each layer, and that sequence is the input for computing other layer values (say the one below and the one above) at the next time step.

Hence the Markov chain is really over the output variable (and associated higher-level hidden layers), and the input sequence only serves to condition that chain, with back-propagation 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.

Zöhrer and Pernkopf (2014) introduced a hybrid model that combines a supervised objective (as in the above work) and an unsupervised objective (as in the original GSN work), by simply adding (with a different weight) the supervised and unsupervised costs i.e., the reconstruction log-probabilities of $\mathbf{y}$ and $\mathbf{x}$ respectively. Such a hybrid criterion had previously been introduced for RBMs by Larochelle and Bengio (2008). They show improved classification performance using this scheme.

20.13 Other Generation Schemes

The methods we have described so far use either MCMC sampling, ancestral sampling, or some mixture of the two to generate samples. While these are the most popular approaches to generative modeling, they are by no means the only approaches.

Sohl-Dickstein *et al.* (2015) developed a **diffusion inversion** training scheme for learning a generative model, based on non-equilibrium thermodynamics. The approach is based on the idea that the probability distributions we wish to sample from have structure. This structure can gradually be destroyed by a diffusion process that incrementally changes the probability distribution to have more entropy. To form a generative model, we can run the process in reverse, by training a model that gradually restores the structure to an unstructured distribution. By iteratively applying a process that brings a distribution closer to the target one, we can gradually approach that target distribution. This approach resembles MCMC methods in the sense that it involves many iterations to produce a sample. However, the model is defined to be the probability distribution produced by the final step of the chain. In this sense, there is no approximation induced by the iterative procedure. The approach introduced by Sohl-Dickstein *et al.* (2015) is also very close to the generative interpretation of the denoising autoencoder

(section 20.11.1). As with the denoising autoencoder, diffusion inversion trains a transition operator that attempts to probabilistically undo the effect of adding some noise. The difference is that diffusion inversion requres undoing only one step of the diffusion process, rather than traveling all the way back to a clean data point. This addresses the following dilemma present with the ordinary reconstruction log-likelihood objective of denoising autoencoders: with small levels of noise the learner only sees configurations near the data points, while with large levels of noise it is asked to do an almost impossible job (because the denoising distribution is highly complex and multi-modal). With the diffusion inversion objective, the learner can learn the shape of the density around the data points more precisely as well as remove spurious modes that could show up far from the data points.

Another approach to sample generation is the **approximate Bayesian computation** (ABC) framework (Rubin *et al.*, 1984). In this approach, samples are rejected or modified in order to make the moments of selected functions of the samples match those of the desired distribution. While this idea uses the moments of the samples like in moment matching, it is different from moment matching because it modifies the samples themselves, rather than training the model to automatically emit samples with the correct moments. Bachman and Precup (2015) showed how to use ideas from ABC in the context of deep learning, by using ABC to shape the MCMC trajectories of GSNs.

We expect that many other possible approaches to generative modeling await discovery.

20.14 Evaluating Generative Models

Researchers studying generative models often need to compare one generative model to another, usually in order to demonstrate that a newly invented generative model is better at capturing some distribution than the pre-existing models.

This can be a difficult and subtle task. In many cases, we can not actually evaluate the log probability of the data under the model, but only an approximation. In these cases, it is important to think and communicate clearly about exactly what is being measured. For example, suppose we can evaluate a stochastic estimate of the log-likelihood for model A, and a deterministic lower bound on the log-likelihood for model B. If model A gets a higher score than model B, which is better? If we care about determining which model has a better internal representation of the distribution, we actually cannot tell, unless we have some way of determining how loose the bound for model B is. However, if we care about how well we can use the model in practice, for example to perform anomaly detection, then it is fair to

say that a model is preferable based on a criterion specific to the practical task of interest, e.g., based on ranking test examples and ranking criteria such as precision and recall.

Another subtlety of evaluating generative models is that the evaluation metrics are often hard research problems in and of themselves. It can be very difficult to establish that models are being compared fairly. For example, suppose we use AIS to estimate $\log Z$ in order to compute $\log \tilde{p}(\boldsymbol{x}) - \log Z$ for a new model we have just invented. A computationally economical implementation of AIS may fail to find several modes of the model distribution and underestimate Z, which will result in us overestimating $\log p(\boldsymbol{x})$. It can thus be difficult to tell whether a high likelihood estimate is due to a good model or a bad AIS implementation.

Other fields of machine learning usually allow for some variation in the preprocessing of the data. For example, when comparing the accuracy of object recognition algorithms, it is usually acceptable to preprocess the input images slightly differently for each algorithm based on what kind of input requirements it has. Generative modeling is different because changes in preprocessing, even very small and subtle ones, are completely unacceptable. Any change to the input data changes the distribution to be captured and fundamentally alters the task. For example, multiplying the input by 0.1 will artificially increase likelihood by a factor of 10.

Issues with preprocessing commonly arise when benchmarking generative models on the MNIST dataset, one of the more popular generative modeling benchmarks. MNIST consists of grayscale images. Some models treat MNIST images as points in a real vector space, while others treat them as binary. Yet others treat the grayscale values as probabilities for a binary samples. It is essential to compare real-valued models only to other real-valued models and binary-valued models only to other binary-valued models. Otherwise the likelihoods measured are not on the same space. For binary-valued models, the log-likelihood can be at most zero, while for real-valued models it can be arbitrarily high, since it is the measurement of a density. Among binary models, it is important to compare models using exactly the same kind of binarization. For example, we might binarize a gray pixel to 0 or 1 by thresholding at 0.5, or by drawing a random sample whose probability of being 1 is given by the gray pixel intensity. If we use the random binarization, we might binarize the whole dataset once, or we might draw a different random example for each step of training and then draw multiple samples for evaluation. Each of these three schemes yields wildly different likelihood numbers, and when comparing different models it is important that both models use the same binarization scheme for training and for evaluation. In fact, researchers who apply a single random

binarization step share a file containing the results of the random binarization, so that there is no difference in results based on different outcomes of the binarization step.

Because being able to generate realistic samples from the data distribution is one of the goals of a generative model, practitioners often evaluate generative models by visually inspecting the samples. In the best case, this is done not by the researchers themselves, but by experimental subjects who do not know the source of the samples (Denton *et al.*, 2015). Unfortunately, it is possible for a very poor probabilistic model to produce very good samples. A common practice to verify if the model only copies some of the training examples is illustrated in figure 16.1. The idea is to show for some of the generated samples their nearest neighbor in the training set, according to Euclidean distance in the space of $\boldsymbol{x}$. This test is intended to detect the case where the model overfits the training set and just reproduces training instances. It is even possible to simultaneously underfit and overfit yet still produce samples that individually look good. Imagine a generative model trained on images of dogs and cats that simply learns to reproduce the training images of dogs. Such a model has clearly overfit, because it does not produces images that were not in the training set, but it has also underfit, because it assigns no probability to the training images of cats. Yet a human observer would judge each individual image of a dog to be high quality. In this simple example, it would be easy for a human observer who can inspect many samples to determine that the cats are absent. In more realistic settings, a generative model trained on data with tens of thousands of modes may ignore a small number of modes, and a human observer would not easily be able to inspect or remember enough images to detect the missing variation.

Since the visual quality of samples is not a reliable guide, we often also evaluate the log-likelihood that the model assigns to the test data, when this is computationally feasible. Unfortunately, in some cases the likelihood seems not to measure any attribute of the model that we really care about. For example, real-valued models of MNIST can obtain arbitrarily high likelihood by assigning arbitrarily low variance to background pixels that never change. Models and algorithms that detect these constant features can reap unlimited rewards, even though this is not a very useful thing to do. The potential to achieve a cost approaching negative infinity is present for any kind of maximum likelihood problem with real values, but it is especially problematic for generative models of MNIST because so many of the output values are trivial to predict. This strongly suggests a need for developing other ways of evaluating generative models.

Theis *et al.* (2015) review many of the issues involved in evaluating generative

models, including many of the ideas described above. They highlight the fact that there are many different uses of generative models and that the choice of metric must match the intended use of the model. For example, some generative models are better at assigning high probability to most realistic points while other generative models are better at rarely assigning high probability to unrealistic points. These differences can result from whether a generative model is designed to minimize $D_{\mathrm{KL}}(p_{\mathrm{data}} \| p_{\mathrm{model}})$ or $D_{\mathrm{KL}}(p_{\mathrm{model}} \| p_{\mathrm{data}})$, as illustrated in figure 3.6. Unfortunately, even when we restrict the use of each metric to the task it is most suited for, all of the metrics currently in use continue to have serious weaknesses. One of the most important research topics in generative modeling is therefore not just how to improve generative models, but in fact, designing new techniques to measure our progress.

20.15 Conclusion

Training generative models with hidden units is a powerful way to make models understand the world represented in the given training data. By learning a model $p_{\mathrm{model}}(\boldsymbol{x})$ and a representation $p_{\mathrm{model}}(\boldsymbol{h} \mid \boldsymbol{x})$, a generative model can provide answers to many inference problems about the relationships between input variables in $\boldsymbol{x}$ and can provide many different ways of representing $\boldsymbol{x}$ by taking expectations of $\boldsymbol{h}$ at different layers of the hierarchy. Generative models hold the promise to provide AI systems with a framework for all of the many different intuitive concepts they need to understand, and the ability to reason about these concepts in the face of uncertainty. We hope that our readers will find new ways to make these approaches more powerful and continue the journey to understanding the principles that underlie learning and intelligence.