watch, even if they did not rate them (this is called **implicit feedback**), and this can be used as useful side information.

Another source of side information concerns the content of the movie, such as the movie genre, the list of the actors, or a synopsis of the plot. This can be denoted by $\mathbf{x}_v$, the features of the video. (In the case where we just have the id of the video, we can treat $\mathbf{x}_v$ as a $|\mathcal{V}|$-dimensional bit vector with just one bit turned on.) We may also know features about the user, which we can denote by $\mathbf{x}_u$. In some cases, we only know if the user clicked on the video or not, that is, we may not have a numerical rating. We can then modify the model as follows:

$$p(R(u,v)|\mathbf{x}_u, \mathbf{x}_v, \boldsymbol{\theta}) = \text{Ber}(R(u,v)|(\mathbf{U}\mathbf{x}_u)^T(\mathbf{V}\mathbf{x}_v)) \tag{27.94}$$

where $\mathbf{U}$ is a $|\mathcal{U}| \times K$ matrix, and $\mathbf{V}$ is a $|\mathcal{V}| \times K$ matrix (we can incorporate an offset term by appending a 1 to $\mathbf{x}_u$ and $\mathbf{x}_v$ in the usual way). A method for computing the approximate posterior $p(\mathbf{U}, \mathbf{V}|\mathcal{D})$ in an online fashion, using ADF and EP, was described in (Stern et al. 2009). This was implemented by Microsoft and has been deployed to predict click through rates on all the ads used by Bing.

Unfortunately, fitting this model just from positive binary data can result in an over prediction of links, since no negative examples are included. Better performance is obtained if one has access to the set of all videos shown to the user, of which at most one was picked; data of this form is known as an **impression log**. In this case, we can use a multinomial model instead of a binary model; in (Yang et al. 2011), this was shown to work much better than a binary model. To understand why, suppose some is presented with a choice of an action movie starring Arnold Schwarzenegger, an action movie starring Vin Diesel, and a comedy starring Hugh Grant. If the user picks Arnold Schwarzenegger, we learn not only that they like prefer action movies to comedies, but also that they prefer Schwarzenegger to Diesel. This is more informative than just knowing that they like Schwarzenegger and action movies.

27.7 Restricted Boltzmann machines (RBMs)

So far, all the models we have proposed in this chapter have been representable by directed graphical models. But some models are better represented using undirected graphs. For example, the **Boltzmann machine** (Ackley et al. 1985) is a pairwise MRF with hidden nodes $\mathbf{h}$ and visible nodes $\mathbf{v}$, as shown in Figure 27.30(a). The main problem with the Boltzmann machine is that exact inference is intractable, and even approximate inference, using e.g., Gibbs sampling, can be slow. However, suppose we restrict the architecture so that the nodes are arranged in layers, and so that there are no connections between nodes within the same layer (see Figure 27.30(b)). Then the model has the form

$$p(\mathbf{h}, \mathbf{v}|\boldsymbol{\theta}) = \frac{1}{Z(\boldsymbol{\theta})} \prod_{r=1}^{R} \prod_{k=1}^{K} \psi_{rk}(v_r, h_k) \tag{27.95}$$

where R is the number of visible (response) variables, K is the number of hidden variables, and $\mathbf{v}$ plays the role of $\mathbf{y}$ earlier in this chapter. This model is known as a **restricted Boltzmann machine** (**RBM**) (Hinton 2002), or a **harmonium** (Smolensky 1986).

An RBM is a special case of a **product of experts** (PoE) (Hinton 1999), which is so-called because we are multiplying together a set of "experts" (here, potential functions on each edge)

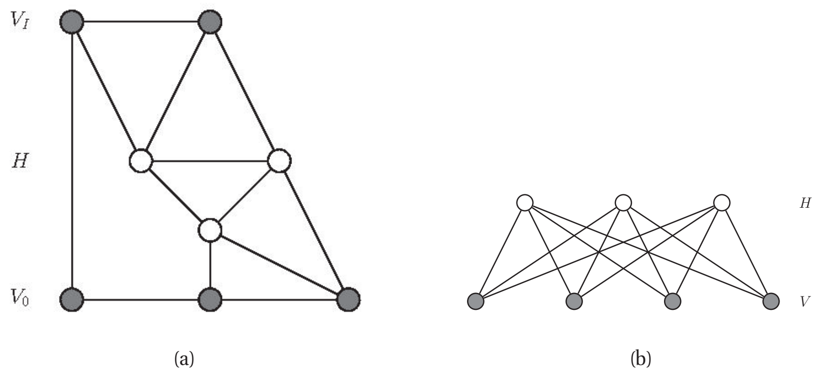


Figure 27.30 (a) A general Boltzmann machine, with an arbitrary graph structure. The shaded (visible) nodes are partitioned into input and output, although the model is actually symmetric and defines a joint density on all the nodes. (b) A restricted Boltzmann machine with a bipartite structure. Note the lack of intra-layer connections.

and then normalizing, whereas in a mixture of experts, we take a convex combination of normalized distributions. The intuitive reason why PoE models might work better than a mixture is that each expert can enforce a constraint (if the expert has a value which is $\gg 1$ or $\ll 1$) or a "don't care" condition (if the expert has value 1). By multiplying these experts together in different ways we can create "sharp" distributions which predict data which satisfies the specified constraints (Hinton and Teh 2001). For example, consider a distributed model of text. A given document might have the topics "government", "mafia" and "playboy". If we "multiply" the predictions of each topic together, the model may give very high probability to the word "Berlusconi"[9] (Salakhutdinov and Hinton 2010). By contrast, adding together experts can only make the distribution broader (see Figure 14.17).

Typically the hidden nodes in an RBM are binary, so $\mathbf{h}$ specifies which constraints are active. It is worth comparing this with the directed models we have discussed. In a mixture model, we have one hidden variable $q \in \{1, \ldots, K\}$. We can represent this using a set of K bits, with the restriction that exactly one bit is on at a time. This is called a **localist encoding**, since only one hidden unit is used to generate the response vector. This is analogous to the hypothetical notion of **grandmother cells** in the brain, that are able to recognize only one kind of object. By contrast, an RBM uses a **distributed encoding**, where many units are involved in generating each output. Models that used vector-valued hidden variables, such as $\boldsymbol{\pi} \in S_K$, as in mPCA/LDA, or $\mathbf{z} \in \mathbb{R}^K$, as in ePCA also use distributed encodings.

The main difference between an RBM and directed two-layer models is that the hidden variables are conditionally independent given the visible variables, so the posterior factorizes:

$$p(\mathbf{h}|\mathbf{v}, \boldsymbol{\theta}) = \prod_k p(h_k|\mathbf{v}, \boldsymbol{\theta}) \tag{27.96}$$

This makes inference much simpler than in a directed model, since we can estimate each h_k

9. Silvio Berlusconi is the current (2011) prime minister of Italy.

Visible	Hidden	Name	Reference
Binary	Binary	Binary RBM	(Hinton 2002)
Gaussian	Binary	Gaussian RBM	(Welling and Sutton 2005)
Categorical	Binary	Categorical RBM	(Salakhutdinov et al. 2007)
Multiple categorical	Binary	Replicated softmax/ undirected LDA	(Salakhutdinov and Hinton 2010)
Gaussian	Gaussian	Undirected PCA	(Marks and Movellan 2001)
Binary	Gaussian	Undirected binary PCA	(Welling and Sutton 2005)

Table 27.2 Summary of different kinds of RBM.

independently and in parallel, as in a feedforward neural network. The disadvantage is that training undirected models is much harder, as we discuss below.

27.7.1 Varieties of RBMs

In this section, we describe various forms of RBMs, by defining different pairwise potential functions. See Table 27.2 for a summary. All of these are special cases of the **exponential family harmonium** (Welling et al. 2004).

27.7.1.1 Binary RBMs

The most common form of RBM has binary hidden nodes and binary visible nodes. The joint distribution then has the following form:

$$\begin{aligned} p(\mathbf{v},\mathbf{h}|\boldsymbol{\theta}) &= \frac{1}{Z(\boldsymbol{\theta})}\exp(-E(\mathbf{v},\mathbf{h};\boldsymbol{\theta})) && (27.97)\\ E(\mathbf{v},\mathbf{h};\boldsymbol{\theta}) &\triangleq -\sum_{r=1}^{R}\sum_{k=1}^{K} v_r h_k W_{rk} - \sum_{r=1}^{R} v_r b_r - \sum_{k=1}^{K} h_k c_k && (27.98)\\ &= -(\mathbf{v}^T\mathbf{W}\mathbf{h} + \mathbf{v}^T\mathbf{b} + \mathbf{h}^T\mathbf{c}) && (27.99)\\ Z(\boldsymbol{\theta}) &= \sum_{\mathbf{v}}\sum_{\mathbf{h}}\exp(-E(\mathbf{v},\mathbf{h};\boldsymbol{\theta})) && (27.100) \end{aligned}$$

where E is the energy function, $\mathbf{W}$ is a $R\times K$ weight matrix, $\mathbf{b}$ are the visible bias terms, $\mathbf{c}$ are the hidden bias terms, and $\boldsymbol{\theta} = (\mathbf{W},\mathbf{b},\mathbf{c})$ are all the parameters. For notational simplicity, we will absorb the bias terms into the weight matrix by clamping dummy units $v_0 = 1$ and $h_0 = 1$ and setting $\mathbf{w}_{0,:} = \mathbf{c}$ and $\mathbf{w}_{:,0} = \mathbf{b}$. Note that naively computing $Z(\boldsymbol{\theta})$ takes $O(2^R 2^K)$ time but we can reduce this to $O(\min\{R2^K, K2^R\})$ time (Exercise 27.1).

When using a binary RBM, the posterior can be computed as follows:

$$p(\mathbf{h}|\mathbf{v},\boldsymbol{\theta}) = \prod_{k=1}^{K} p(h_k|\mathbf{v},\boldsymbol{\theta}) = \prod_k \mathrm{Ber}(h_k|\mathrm{sigm}(\mathbf{w}_{:,k}^T\mathbf{v})) \quad (27.101)$$

By symmetry, one can show that we can generate data given the hidden variables as follows:

$$p(\mathbf{v}|\mathbf{h},\boldsymbol{\theta}) = \prod_r p(v_r|\mathbf{h},\boldsymbol{\theta}) = \prod_r \mathrm{Ber}(v_r|\mathrm{sigm}(\mathbf{w}_{r,:}^T\mathbf{h})) \quad (27.102)$$

We can write this in matrix-vetor notation as follows:

$$\mathbb{E}\left[\mathbf{h}|\mathbf{v}\boldsymbol{\theta}\right] = \text{sigm}(\mathbf{W}^T\mathbf{v}) \quad (27.103)$$

$$\mathbb{E}\left[\mathbf{v}|\mathbf{h},\boldsymbol{\theta}\right] = \text{sigm}(\mathbf{W}\mathbf{h}) \quad (27.104)$$

The weights in $\mathbf{W}$ are called the **generative weights**, since they are used to generate the observations, and the weights in $\mathbf{W}^T$ are called the **recognition weights**, since they are used to recognize the input.

From Equation 27.101, we see that we activate hidden node k in proportion to how much the input vector $\mathbf{v}$ "looks like" the weight vector $\mathbf{w}_{:,k}$ (up to scaling factors). Thus each hidden node captures certain features of the input, as encoded in its weight vector, similar to a feedforward neural network.

27.7.1.2 Categorical RBM

We can extend the binary RBM to categorical visible variables by using a 1-of-C encoding, where C is the number of states for each v_{ir}. We define a new energy function as follows (Salakhutdinov et al. 2007; Salakhutdinov and Hinton 2010):

$$E(\mathbf{v},\mathbf{h};\boldsymbol{\theta}) \triangleq -\sum_{r=1}^{R}\sum_{k=1}^{K}\sum_{c=1}^{C} v_r^c h_k W_{rk}^c - \sum_{r=1}^{R}\sum_{c=1}^{C} v_r^c b_r^c - \sum_{k=1}^{K} h_k c_k \quad (27.105)$$

The full conditionals are given by

$$p(v_r|\mathbf{h},\boldsymbol{\theta}) = \text{Cat}(\mathcal{S}(\{b_r^c + \sum_k h_k W_{rk}^c\}_{c=1}^C)) \quad (27.106)$$

$$p(h_k = 1|\mathbf{c},\boldsymbol{\theta}) = \text{sigm}(c_k + \sum_r \sum_c v_r^c W_{rk}^c) \quad (27.107)$$

27.7.1.3 Gaussian RBM

We can generalize the model to handle real-valued data. In particular, a **Gaussian RBM** has the following energy function:

$$E(\mathbf{v},\mathbf{h}|\boldsymbol{\theta}) = -\sum_{r=1}^{R}\sum_{k=1}^{K} W_{rk} h_k v_r - \frac{1}{2}\sum_{r=1}^{R}(v_r - b_r)^2 - \sum_{k=1}^{K} a_k h_k \quad (27.108)$$

The parameters of the model are $\boldsymbol{\theta} = (w_{rk}, a_k, b_r)$. (We have assumed the data is standardized, so we fix the variance to $\sigma^2 = 1$.) Compare this to a Gaussian in information form:

$$\mathcal{N}_c(\mathbf{v}|\boldsymbol{\eta},\boldsymbol{\Lambda}) \propto \exp(\boldsymbol{\eta}^T\mathbf{v} - \frac{1}{2}\mathbf{v}^T\boldsymbol{\Lambda}\mathbf{v}) \quad (27.109)$$

where $\boldsymbol{\eta} = \boldsymbol{\Lambda}\boldsymbol{\mu}$. We see that we have set $\boldsymbol{\Lambda} = \mathbf{I}$, and $\boldsymbol{\eta} = \sum_k h_k \mathbf{w}_{:,k}$. Thus the mean is given by $\boldsymbol{\mu} = \boldsymbol{\Lambda}^{-1}\boldsymbol{\eta} = \sum_k h_k \mathbf{w}_{:,k}$. The full conditionals, which are needed for inference and learning, are given by

$$p(v_r|\mathbf{h}, \boldsymbol{\theta}) = \mathcal{N}(v_r|b_r + \sum_k w_{rk} h_k, 1) \tag{27.110}$$

$$p(h_k = 1|\mathbf{v}, \boldsymbol{\theta}) = \text{sigm}\left(c_k + \sum_r w_{rk} v_r\right) \tag{27.111}$$

We see that each visible unit has a Gaussian distribution whose mean is a function of the hidden bit vector. More powerful models, which make the (co)variance depend on the hidden state, can also be developed (Ranzato and Hinton 2010).

27.7.1.4 RBMs with Gaussian hidden units

If we use Gaussian latent variables and Gaussian visible variables, we get an undirected version of factor analysis. However, it turns out that it is identical to the standard directed version (Marks and Movellan 2001).

If we use Gaussian latent variables and categorical observed variables, we get an undirected version of categorical PCA (Section 27.2.2). In (Salakhutdinov et al. 2007), this was applied to the Netflix collaborative filtering problem, but was found to be significantly inferior to using binary latent variables, which have more expressive power.

27.7.2 Learning RBMs

In this section, we discuss some ways to compute ML parameter estimates of RBMs, using gradient-based optimizers. It is common to use stochastic gradient descent, since RBMs often have many parameters and therefore need to be trained on very large datasets. In addition, it is standard to use ℓ_2 regularization, a technique that is often called weight decay in this context. This requires a very small change to the objective and gradient, as discussed in Section 8.3.6.

27.7.2.1 Deriving the gradient using $p(\mathbf{h}, \mathbf{v}|\boldsymbol{\theta})$

To compute the gradient, we can modify the equations from Section 19.5.2, which show how to fit a generic latent variable maxent model. In the context of the Boltzmann machine, we have one feature per edge, so the gradient is given by

$$\frac{\partial \ell}{\partial w_{rk}} = \frac{1}{N} \sum_{i=1}^{N} \mathbb{E}\left[v_r h_k | \mathbf{v}_i, \boldsymbol{\theta}\right] - \mathbb{E}\left[v_r h_k | \boldsymbol{\theta}\right] \tag{27.112}$$

We can write this in matrix-vector form as follows:

$$\nabla_{\mathbf{w}}\, \ell = \mathbb{E}_{p_{\text{emp}}(\cdot|\boldsymbol{\theta})}\left[\mathbf{v}\mathbf{h}^T\right] - \mathbb{E}_{p(\cdot|\boldsymbol{\theta})}\left[\mathbf{v}\mathbf{h}^T\right] \tag{27.113}$$

where $p_{\text{emp}}(\mathbf{v}, \mathbf{h}|\boldsymbol{\theta}) \triangleq p(\mathbf{h}|\mathbf{v}, \boldsymbol{\theta}) p_{\text{emp}}(\mathbf{v})$, and $p_{\text{emp}}(\mathbf{v}) = \frac{1}{N}\sum_{i=1}^{N} \delta_{\mathbf{v}_i}(\mathbf{v})$ is the empirical distribution. (We can derive a similar expression for the bias terms by setting $v_r = 1$ or $h_k = 1$.)

The first term on the gradient, when $\mathbf{v}$ is fixed to a data case, is sometimes called the **clamped phase**, and the second term, when $\mathbf{v}$ is free, is sometimes called the **unclamped**

phase. When the model expectations match the empirical expectations, the two terms cancel out, the gradient becomes zero and learning stops. This algorithm was first proposed in (Ackley et al. 1985). The main problem is efficiently computing the expectations. We discuss some ways to do this below.

27.7.2.2 Deriving the gradient using $p(\mathbf{v}|\boldsymbol{\theta})$

We now present an alternative way to derive Equation 27.112, which also applies to other energy based models. First we marginalize out the hidden variables and write the RBM in the form $p(\mathbf{v}|\boldsymbol{\theta}) = \frac{1}{Z(\boldsymbol{\theta})}\exp(-F(\mathbf{v};\boldsymbol{\theta}))$, where $F(\mathbf{v};\boldsymbol{\theta})$ is the **free energy**:

$$F(\mathbf{v}) \triangleq \sum_{\mathbf{h}} E(\mathbf{v},\mathbf{h}) = \sum_{\mathbf{h}} \exp\left(\sum_{r=1}^{R}\sum_{k=1}^{K} v_r h_k W_{rk}\right) \tag{27.114}$$

$$= \sum_{\mathbf{h}} \prod_{k=1}^{K} \exp\left(\sum_{r=1}^{R} v_r h_k W_{rk}\right) \tag{27.115}$$

$$= \prod_{k=1}^{K} \sum_{h_r\in\{0,1\}} \exp\left(\sum_{r=1}^{R} v_r h_r W_{rk}\right) \tag{27.116}$$

$$= \prod_{k=1}^{K} \left(1+\exp(\sum_{r=1}^{R} v_r W_{rk})\right) \tag{27.117}$$

Next we write the (scaled) log-likelihood in the following form:

$$\ell(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N}\log p(\mathbf{v}_i|\boldsymbol{\theta}) = -\frac{1}{N}\sum_{i=1}^{N} F(\mathbf{v}_i|\boldsymbol{\theta}) - \log Z(\boldsymbol{\theta}) \tag{27.118}$$

Using the fact that $Z(\boldsymbol{\theta}) = \sum_{\mathbf{v}}\exp(-F(\mathbf{v};\boldsymbol{\theta}))$ we have

$$\nabla\ell(\boldsymbol{\theta}) = -\frac{1}{N}\sum_{i=1}^{N}\nabla F(\mathbf{v}_i) - \frac{\nabla Z}{Z} \tag{27.119}$$

$$= -\frac{1}{N}\sum_{i=1}^{N}\nabla F(\mathbf{v}_i) + \sum_{\mathbf{v}}\nabla F(\mathbf{v})\frac{\exp(-F(\mathbf{v}))}{Z} \tag{27.120}$$

$$= -\frac{1}{N}\sum_{i=1}^{N}\nabla F(\mathbf{v}_i) + \mathbb{E}\left[\nabla F(\mathbf{v})\right] \tag{27.121}$$

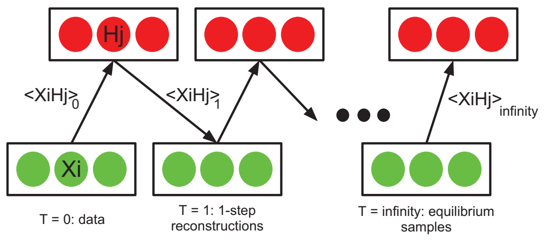
Plugging in the free energy (Equation 27.117), one can show that

$$\frac{\partial}{\partial w_{rk}}F(\mathbf{v}) = -v_r\mathbb{E}\left[h_k|\mathbf{v},\boldsymbol{\theta}\right] = -\mathbb{E}\left[v_r h_k|\mathbf{v},\boldsymbol{\theta}\right] \tag{27.122}$$

Hence

$$\frac{\partial}{\partial w_{rk}}\ell(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N}\mathbb{E}\left[v_r h_k|\mathbf{v},\boldsymbol{\theta}\right] - \mathbb{E}\left[v_r h_k|\boldsymbol{\theta}\right] \tag{27.123}$$

which matches Equation 27.112.



Hj
<XiHj>0
<XiHj>1
<XiHj>infinity
Xi
T = 0: data
T = 1: 1-step reconstructions
T = infinity: equilibrium samples

Figure 27.31 Illustration of Gibbs sampling in an RBM. The visible nodes are initialized at a datavector, then we sample a hidden vector, then another visible vector, etc. Eventually (at "infinity") we will be producing samples from the joint distribution $p(\mathbf{v}, \mathbf{h}|\boldsymbol{\theta})$.

27.7.2.3 Approximating the expectations

We can approximate the expectations needed to evaluate the gradient by performing block Gibbs sampling, using Equations 27.101 and 27.102. In more detail, we can sample from the joint distribution $p(\mathbf{v}, \mathbf{h}|\boldsymbol{\theta})$ as follows: initialize the chain at vv^1 (e.g. by setting $\mathbf{v}^1 = \mathbf{v}_i$ for some data vector), and then sample from $\mathbf{h}^1 \sim p(\mathbf{h}|\mathbf{v}^1)$, then from $\mathbf{v}^2 \sim p(\mathbf{v}|\mathbf{h}^1)$, then from $\mathbf{h}^2 \sim p(\mathbf{h}|\mathbf{v}^2)$, etc. See Figure 27.31 for an illustration. Note, however, that we have to wait until the Markov chain reaches equilibrium (i.e., until it has "burned in") before we can interpret the samples as coming from the joint distribution of interest, and this might take a long time.

A faster alternative is to use mean field, where we make the approximation $\mathbb{E}[v_r h_k] \approx \mathbb{E}[v_r]\,\mathbb{E}[h_k]$. However, since $p(\mathbf{v}, \mathbf{h})$ is typically multimodal, this is usually a very poor approximation, since it will average over different modes (see Section 21.2.2). Furthermore, there is a more subtle reason not to use mean field: since the gradient has the form $\mathbb{E}[v_r h_k|\mathbf{v}] - \mathbb{E}[v_r h_k]$, we see that the negative sign in front means that the method will try to make the variational bound as loose as possible (Salakhutdinov and Hinton 2009). This explains why earlier attempts to use mean field to learn Boltzmann machines (e.g., (Kappen and Rodriguez 1998)) did not work.

27.7.2.4 Contrastive divergence

The problem with using Gibbs sampling to compute the gradient is that it is slow. We now present a faster method known as **contrastive divergence** or **CD** (Hinton 2002). CD was originally derived by approximating an objective function defined as the difference of two KL divergences, rather than trying to maximize the likelihood itself. However, from an algorithmic point of view, it can be thought of as similar to stochastic gradient descent, except it approximates the "unclamped" expectations with "brief" Gibbs sampling where we initialize each Markov chain at the data vectors. That is, we approximate the gradient for one datavector as follows:

$$\nabla_{\mathbf{w}}\, \ell \approx \mathbb{E}\left[\mathbf{v}\mathbf{h}^T|\mathbf{v}_i\right] - \mathbb{E}_q\left[\mathbf{v}\mathbf{h}^T\right] \tag{27.124}$$

where q corresponds to the distribution generated by K up-down Gibbs sweeps, started at $\mathbf{v}_i$, as in Figure 27.31. This is known as CD-K. In more detail, the procedure (for $K = 1$) is as follows:

$$\mathbf{h}_i \sim p(\mathbf{h}|\mathbf{v}_i, \boldsymbol{\theta}) \tag{27.125}$$

$$\mathbf{v}'_i \sim p(\mathbf{v}|\mathbf{h}_i, \boldsymbol{\theta}) \tag{27.126}$$

$$\mathbf{h}'_i \sim p(\mathbf{h}|\mathbf{v}'_i, \boldsymbol{\theta}) \tag{27.127}$$

We then make the approximation

$$\mathbb{E}_q\left[\mathbf{v}\mathbf{h}^T\right] \approx \mathbf{v}_i(\mathbf{h}'_i)^T \tag{27.128}$$

Such samples are sometimes called **fantasy data**. We can think of $\mathbf{v}'_i$ as the model's best attempt at reconstructing $\mathbf{v}_i$ after being coded and then decoded by the model. This is similar to the way we train auto-encoders, which are models which try to "squeeze" the data through a restricted parametric "bottleneck" (see Section 28.3.2).

In practice, it is common to use $\mathbb{E}[\mathbf{h}|\mathbf{v}'_i]$ instead of a sampled value $\mathbf{h}'_i$ in the final upwards pass, since this reduces the variance. However, it is not valid to use $\mathbb{E}[\mathbf{h}|\mathbf{v}_i]$ instead of sampling $\mathbf{h}_i \sim p(\mathbf{h}|\mathbf{v}_i)$ in the earlier upwards passes, because then each hidden unit would be able to pass more than 1 bit of information, so it would not act as much of a bottleneck.

The whole procedure is summarized in Algorithm 3. (Note that we follow the positive gradient since we are maximizing likelihood.) Various tricks can be used to speed this algorithm up, such as using a momentum term (Section 8.3.2), using mini-batches, averaging the updates, etc. Such details can be found in (Hinton 2010; Swersky et al. 2010).

Algorithm 27.3: CD-1 training for an RBM with binary hidden and visible units

```
1  Initialize weights $\mathbf{W} \in \mathbb{R}^{R \times K}$ randomly;
2  $t := 0$;
3  for each epoch do
4      $t := t + 1$ ;
5      for each minibatch of size $B$ do
6          Set minibatch gradient to zero, $\mathbf{g} := \mathbf{0}$ ;
7          for each case $\mathbf{v}_i$ in the minibatch do
8              Compute $\boldsymbol{\mu}_i = \mathbb{E}[\mathbf{h}|\mathbf{v}_i, \mathbf{W}]$;
9              Sample $\mathbf{h}_i \sim p(\mathbf{h}|\mathbf{v}_i, \mathbf{W})$;
10             Sample $\mathbf{v}'_i \sim p(\mathbf{v}|\mathbf{h}_i, \mathbf{W})$;
11             Compute $\boldsymbol{\mu}'_i = \mathbb{E}[\mathbf{h}|\mathbf{v}'_i, \mathbf{W}]$;
12             Compute gradient $\nabla_{\mathbf{W}} = (\mathbf{v}_i)(\boldsymbol{\mu}_i)^T - (\mathbf{v}'_i)(\boldsymbol{\mu}'_i)^T$ ;
13             Accumulate $\mathbf{g} := \mathbf{g} + \nabla_{\mathbf{W}}$;
14         Update parameters $\mathbf{W} := \mathbf{W} + (\alpha_t/B)\mathbf{g}$
```

27.7.2.5 Persistent CD

In Section 19.5.5, we presented a technique called stochastic maximum likelihood (SML) for fitting maxent models. This avoids the need to run MCMC to convergence at each iteration, by exploiting the fact that the parameters are changing slowly, so the Markov chains will not be pushed too far from equilibrium after each update (Younes 1989). In other words, there are two dynamical processes running at different time scales: the states change quickly, and the parameters change slowly. This algorithm was independently rediscovered in (Tieleman 2008), who called it **persistent CD**. See Algorithm 3 for the pseudocode.

PCD often works better than CD (see e.g., (Tieleman 2008; Marlin et al. 2010; Swersky et al. 2010)), although CD can be faster in the early stages of learning.

Algorithm 27.4: Persistent CD for training an RBM with binary hidden and visible units

```
1  Initialize weights $\mathbf{W} \in \mathbb{R}^{D \times L}$ randomly;
2  Initialize chains $(\mathbf{v}_s, \mathbf{h}_s)_{s=1}^S$ randomly ;
3  for $t = 1, 2, \ldots$ do
4      // Mean field updates ;
5      for each case $i = 1 : N$ do
6          $\mu_{ik} = \text{sigm}(\mathbf{v}_i^T \mathbf{w}_{:,k})$
7      // MCMC updates ;
8      for each sample $s = 1 : S$ do
9          Generate $(\mathbf{v}_s, \mathbf{h}_s)$ by brief Gibbs sampling from old $(\mathbf{v}_s, \mathbf{h}_s)$
10     // Parameter updates ;
11     $\mathbf{g} = \frac{1}{N} \sum_{i=1}^N \mathbf{v}_i (\boldsymbol{\mu}_i)^T - \frac{1}{S} \sum_{s=1}^S \mathbf{v}_s (\mathbf{h}_s)^T$ ;
12     $\mathbf{W} := \mathbf{W} + \alpha_t \mathbf{g}$;
13     Decrease $\alpha_t$
```

27.7.3 Applications of RBMs

The main application of RBMs is as a building block for deep generative models, which we discuss in Section 28.2. But they can also be used as substitutes for directed two-layer models. They are particularly useful in cases where inference of the hidden states at test time must be fast. We give some examples below.

27.7.3.1 Language modeling and document retrieval

We can use a categorical RBM to define a generative model for bag-of-words, as an alternative to LDA. One subtlety is that the partition function in an undirected models depends on how big the graph is, and therefore on how long the document is. A solution to this was proposed in (Salakhutdinov and Hinton 2010): use a categorical RBM with tied weights, but multiply the hidden activation bias terms c_k by the document length L to compensate form the fact that the observed word-count vector $\mathbf{v}$ is larger in magnitude:

$$E(\mathbf{v}, \mathbf{h}; \boldsymbol{\theta}) \triangleq -\sum_{k=1}^K \sum_{c=1}^C v^c h_k W_k^c - \sum_{c=1}^C v^c b_r^c - L \sum_{k=1}^K h_k c_k \quad (27.129)$$

Data set	Number of docs		K	$\bar{D}$	St. Dev.	Avg. Test perplexity per word (in nats)			
	Train	Test				LDA-50	LDA-200	R. Soft-50	Unigram
NIPS	1,690	50	13,649	98.0	245.3	3576	3391	3405	4385
20-news	11,314	7,531	2,000	51.8	70.8	1091	1058	953	1335
Reuters	794,414	10,000	10,000	94.6	69.3	1437	1142	988	2208

Figure 27.32 Comparison of RBM (replicated softmax) and LDA on three corpora. K is the number of words in the vocabulary, $\overline{D}$ is the average document length, and St. Dev. is the standard deviation of the document length. Source: (Salakhutdinov and Hinton 2010) .

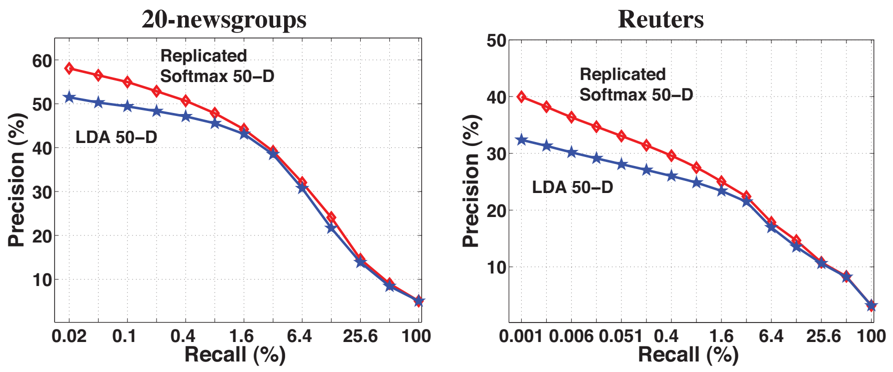


Figure 27.33 Precision-recall curves for RBM (replicated softmax) and LDA on two corpora. From Figure 3 of (Salakhutdinov and Hinton 2010). Used with kind permission of Ruslan Salakhutdinov.

where $v^c = \sum_{l=1}^{L} \mathbb{I}(y_{il} = c)$. This is like having a single multinomial node (so we have dropped the r subscript) with C states, where C is the number of words in the vocabulary. This is called the **replicated softmax model** (Salakhutdinov and Hinton 2010), and is an undirected alternative to mPCA/ LDA.

We can compare the modeling power of RBMs vs LDA by measuring the perplexity on a test set. This can be approximated using annealing importance sampling (Section 24.6.2). The results are shown in Figure 27.32. We see that the LDA is significantly better than a unigram model, but that an RBM is significantly better than LDA.

Another advantage of the LDA is that inference is fast and exact: just a single matrix-vector multiply followed by a sigmoid nonlinearity, as in Equation 27.107. In addition to being faster, the RBM is more accurate. This is illustrated in Figure 27.33, which shows precision-recall curves for RBMs and LDA on two different corpora. These curves were generated as follows: a query document from the test set is taken, its similarity to all the training documents is computed, where the similarity is defined as the cosine of the angle between the two topic vectors, and then the top M documents are returned for varying M. A retrieved document is considered relevant if it has the same class label as that of the query's (this is the only place where labels are used).

27.7.3.2 RBMs for collaborative filtering

RBMs have been applied to the Netflix collaborative filtering competition (Salakhutdinov et al. 2007). In fact, an RBM with binary hidden nodes and categorical visible nodes can slightly outperform SVD. By combining the two methods, performance can be further improved. (The winning entry in the challenge was an ensemble of many different types of model (Koren 2009a).)

Exercises

Exercise 27.1 Partition function for an RBM

Show how to compute $Z(\boldsymbol{\theta})$ for an RBM with K binary hidden nodes and R binary observed nodes in $O(R2^K)$ time, assuming $K < R$.

28 *Deep learning*

28.1 Introduction

Many of the models we have looked at in this book have a simple two-layer architecture of the form $\mathbf{z} \rightarrow \mathbf{y}$ for unsupervised latent variable models, or $\mathbf{x} \rightarrow \mathbf{y}$ for supervised models. However, when we look at the brain, we seem many levels of processing. It is believed that each level is learning features or representations at increasing levels of abstraction. For example, the **standard model** of the visual cortex (Hubel and Wiesel 1962; Serre et al. 2005; Ranzato et al. 2007) suggests that (roughly speaking) the brain first extracts edges, then patches, then surfaces, then objects, etc. (See e.g., (Palmer 1999; Kandel et al. 2000) for more information about how the brain might perform vision.)

This observation has inspired a recent trend in machine learning known as **deep learning** (see e.g., (Bengio 2009), `deeplearning.net`, and the references therein), which attempts to replicate this kind of architecture in a computer. (Note the idea can be applied to non-vision problems as well, such as speech and language.)

In this chapter, we give a brief overview of this new field. However, we caution the reader that the topic of deep learning is currently evolving very quickly, so the material in this chapter may soon be outdated.

28.2 Deep generative models

Deep models often have millions of parameters. Acquiring enough labeled data to train such models is diffcult, despite crowd sourcing sites such as Mechanical Turk. In simple settings, such as hand-written character recognition, it is possible to generate lots of labeled data by making modified copies of a small manually labeled training set (see e.g., Figure 16.13), but it seems unlikely that this approach will scale to complex scenes.[1]

To overcome the problem of needing labeled training data, we will focus on unsupervised learning. The most natural way to perform this is to use generative models. In this section, we discuss three different kinds of deep generative models: directed, undirected, and mixed.

1. There have been some attempts to use computer graphics and video games to generate realistic-looking images of complex scenes, and then to use this as training data for computer vision systems. However, often graphics programs cut corners in order to make perceptually appealing images which are not reflective of the natural statistics of real-world images.

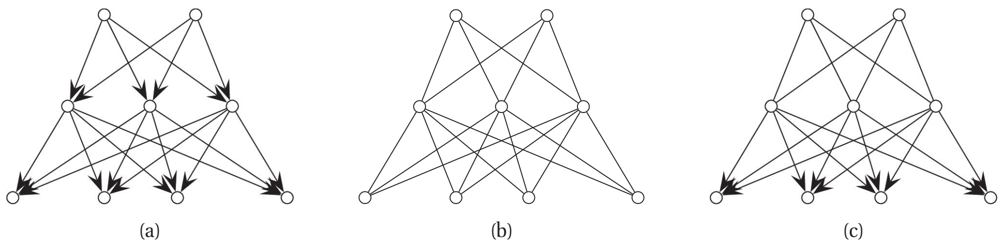


Figure 28.1 Some deep multi-layer graphical models. Observed variables are at the bottom. (a) A directed model. (b) An undirected model (deep Boltzmann machine). (c) A mixed directed-undirected model (deep belief net).

28.2.1 Deep directed networks

Perhaps the most natural way to build a deep generative model is to construct a deep directed graphical model, as shown in Figure 28.1(a). The bottom level contains the observed pixels (or whatever the data is), and the remaining layers are hidden. We have assumed just 3 layers for notational simplicity. The number and size of layers is usually chosen by hand, although one can also use non-parametric Bayesian methods (Adams et al. 2010) or boosting (Chen et al. 2010) to infer the model structure.

We shall call models of this form **deep directed networks** or DDNs. If all the nodes are binary, and all CPDs are logistic functions, this is called a **sigmoid belief net** (Neal 1992). In this case, the model defines the following joint distribution:

$$p(\mathbf{h}_1, \mathbf{h}_2, \mathbf{h}_3, \mathbf{v}|\boldsymbol{\theta}) = \prod_i \text{Ber}(v_i|\text{sigm}(\mathbf{h}_1^T \mathbf{w}_{0i})) \prod_j \text{Ber}(h_{1j}|\text{sigm}(\mathbf{h}_2^T \mathbf{w}_{1j})) \tag{28.1}$$

$$\prod_k \text{Ber}(h_{2k}|\text{sigm}(\mathbf{h}_3^T \mathbf{w}_{2k})) \prod_l \text{Ber}(h_{3l}|w_{3l}) \tag{28.2}$$

Unfortunately, inference in directed models such as these is intractable because the posterior on the hidden nodes is correlated due to explaining away. One can use fast mean field approximations (Jaakkola and Jordan 1996a; Saul and Jordan 2000), but these may not be very accurate, since they approximate the correlated posterior with a factorial posterior. One can also use MCMC inference (Neal 1992; Adams et al. 2010), but this can be quite slow because the variables are highly correlated. Slow inference also results in slow learning.

28.2.2 Deep Boltzmann machines

A natural alternative to a directed model is to construct a deep undirected model. For example, we can stack a series of RBMs on top of each other, as shown in Figure 28.1(b). This is known as a **deep Boltzmann machine** or **DBM** (Salakhutdinov and Hinton 2009). If we have 3 hidden layers, the model is defined as follows:

$$p(\mathbf{h}_1, \mathbf{h}_2, \mathbf{h}_3, \mathbf{v}|\boldsymbol{\theta}) = \frac{1}{Z(\boldsymbol{\theta})} \exp\left(\sum_{ij} v_i h_{1j} W_{1ij} + \sum_{jk} h_{1j} h_{2j} W_{2jk} + \sum_{kl} h_{2k} h_{3l} W_{3kl}\right) \tag{28.3}$$

where we are ignoring constant offset or bias terms.

The main advantage over the directed model is that one can perform efficient block (layer-wise) Gibbs sampling, or block mean field, since all the nodes in each layer are conditionally independent of each other given the layers above and below (Salakhutdinov and Larochelle 2010). The main disadvantage is that training undirected models is more difficult, because of the partition function. However, below we will see a greedy layer-wise strategy for learning deep undirected models.

28.2.3 Deep belief networks

An interesting compromise is to use a model that is partially directed and partially undirected. In particular, suppose we construct a layered model which has directed arrows, except at the top, where there is an undirected bipartite graph, as shown in Figure 28.1(c). This model is known as a **deep belief network** (Hinton et al. 2006) or **DBN**.[2] If we have 3 hidden layers, the model is defined as follows:

$$p(\mathbf{h}_1, \mathbf{h}_2, \mathbf{h}_3, \mathbf{v}|\boldsymbol{\theta}) = \prod_i \text{Ber}(v_i|\text{sigm}(\mathbf{h}_1^T \mathbf{w}_{1i}) \prod_j \text{Ber}(h_{1j}|\text{sigm}(\mathbf{h}_2^T \mathbf{w}_{2j}) \tag{28.4}$$

$$\frac{1}{Z(\boldsymbol{\theta})} \exp\left(\sum_{kl} h_{2k} h_{3l} W_{3kl}\right) \tag{28.5}$$

Essentially the top two layers act as an associative memory, and the remaining layers then generate the output.

The advantage of this peculiar architecture is that we can infer the hidden states in a fast, bottom-up fashion. To see why, suppose we only have two hidden layers, and that $\mathbf{W}_2 = \mathbf{W}_1^T$, so the second level weights are tied to the first level weights (see Figure 28.2(a)). This defines a model of the form $p(\mathbf{h}_1, \mathbf{h}_2, \mathbf{v}|\mathbf{W}_1)$. One can show that the distribution $p(\mathbf{h}_1, \mathbf{v}|\mathbf{W}_1) = \sum_{\mathbf{h}_2} p(\mathbf{h}_1, \mathbf{h}_2, \mathbf{v}|\mathbf{W}_1)$ has the form $p(\mathbf{h}_1, \mathbf{v}|\mathbf{W}_1) = \frac{1}{Z(\mathbf{W}_1)} \exp(\mathbf{v}^T \mathbf{W}_1 \mathbf{h}_1)$, which is equivalent to an RBM. Since the DBN is equivalent to the RBM as far as $p(\mathbf{h}_1, \mathbf{v}|\mathbf{W}_1)$ is concerned, we can infer the posterior $p(\mathbf{h}_1|\mathbf{v}, \mathbf{W}_1)$ in the DBN exactly as in the RBM. This posterior is exact, even though it is fully factorized.

Now the only way to get a factored posterior is if the prior $p(\mathbf{h}_1|\mathbf{W}_1)$ is a **complementary prior**. This is a prior which, when multiplied by the likelihood $p(\mathbf{v}|\mathbf{h}_1)$, results in a perfectly factored posterior. Thus we see that the top level RBM in a DBN acts as a complementary prior for the bottom level directed sigmoidal likelihood function.

If we have multiple hidden levels, and/or if the weights are not tied, the correspondence between the DBN and the RBM does not hold exactly any more, but we can still use the factored inference rule as a form of approximate bottom-up inference. Below we show that this is a valid variational lower bound. This bound also suggests a layer-wise training strategy, that we will explain in more detail later. Note, however, that top-down inference in a DBN is not tractable, so DBNs are usually only used in a feedforward manner.

2. Unforuntately the acronym "DBN" also stands for "dynamic Bayes net" (Section 17.6.7). Geoff Hinton, who invented deep belief networks, has suggested the acronyms **DeeBNs** and **DyBNs** for these two different meanings. However, this terminology is non-standard.

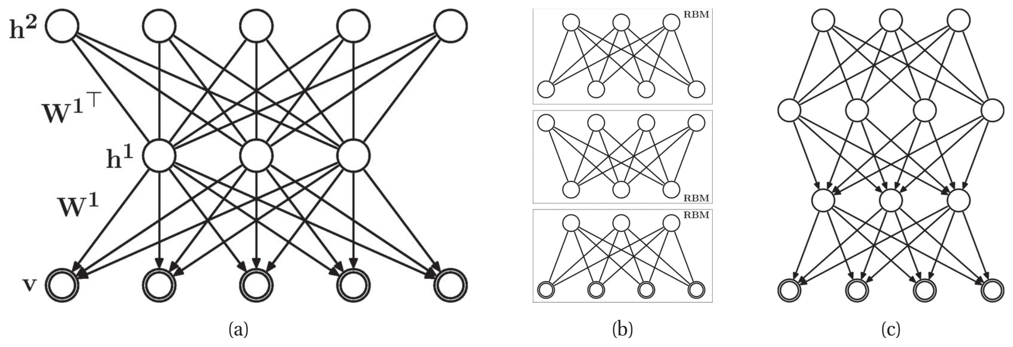


Figure 28.2 (a) A DBN with two hidden layers and tied weights that is equivalent to an RBM. Source: Figure 2.2 of (Salakhutdinov 2009). (b) A stack of RBMs trained greedily. (c) The corresponding DBN. Source: Figure 2.3 of (Salakhutdinov 2009). Used with kind permission of Ruslan Salakhutdinov.

28.2.4 Greedy layer-wise learning of DBNs

The equivalence between DBNs and RBMs suggests the following strategy for learning a DBN.

- Fit an RBM to learn $\mathbf{W}_1$ using methods described in Section 27.7.2.
- Unroll the RBM into a DBN with 2 hidden layers, as in Figure 28.2(a). Now "freeze" the directed weights $\mathbf{W}_1$ and let $\mathbf{W}_2$ be "untied" so it is no longer forced to be equal to $\mathbf{W}_1^T$. We will now learn a better prior for $p(\mathbf{h}_1|\mathbf{W}_2)$ by fitting a second RBM. The input data to this new RBM is the activation of the hidden units $\mathbb{E}[\mathbf{h}_1|\mathbf{v}, \mathbf{W}_1]$ which can be computed using a factorial approximation.
- Continue to add more hidden layers until some stopping criterion is satisified, e.g., you run out of time or memory, or you start to overfit the validation set. Construct the DBN from these RBMs, as illustrated in Figure 28.2(c).

One can show (Hinton et al. 2006) that this procedure always increases a lower bound the observed data likelihood. Of course this procedure might result in overfitting, but that is a different matter.

In practice, we want to be able to use any number of hidden units in each level. This means we will not be able to initialize the weights so that $\mathbf{W}_\ell = \mathbf{W}_{\ell-1}^T$. This voids the theoretical guarantee. Nevertheless the method works well in practice, as we will see. The method can also be extended to train DBMs in a greedy way (Salakhutdinov and Larochelle 2010).

After using the greedy layer-wise training strategy, it is standard to "fine tune" the weights, using a technique called **backfitting**. This works as follows. Perform an upwards sampling pass to the top. Then perform brief Gibbs sampling in the top level RBM, and perform a CD update of the RBM parameters. Finally, perform a downwards ancestral sampling pass (which is an approximate sample from the posterior), and update the logistic CPD parameters using a small gradient step. This is called the **up-down** procedure (Hinton et al. 2006). Unfortunately this procedure is very slow.

28.3 Deep neural networks

Given that DBNs are often only used in a feed-forward, or bottom-up, mode, they are effectively acting like neural networks. In view of this, it is natural to dispense with the generative story and try to fit deep neural networks directly, as we discuss below. The resulting training methods are often simpler to implement, and can be faster.

Note, however, that performance with deep neural nets is sometimes not as good as with probabilistic models (Bengio et al. 2007). One reason for this is that probabilistic models support top-down inference as well as bottom-up inference. (DBNs do not support efficient top-down inference, but DBMs do, and this has been shown to help (Salakhutdinov and Larochelle 2010).) Top-down inference is useful when there is a lot of ambiguity about the correct interpretation of the signal.

It is interesting to note that in the mammalian visual cortex, there are many more feedback connections than there are feedforward connections (see e.g., (Palmer 1999; Kandel et al. 2000)). The role of these feedback connections is not precisely understood, but they presumably provide contextual prior information (e.g., coming from the previous “frame” or retinal glance) which can be used to disambiguate the current bottom-up signals (Lee and Mumford 2003).

Of course, we can simulate the effect of top-down inference using a neural network. However the models we discuss below do not do this.

28.3.1 Deep multi-layer perceptrons

Many decision problems can be reduced to classification, e.g., predict which object (if any) is present in an image patch, or predict which phoneme is present in a given acoustic feature vector. We can solve such problems by creating a deep feedforward neural network or multi-layer perceptron (MLP), as in Section 16.5, and then fitting the parameters using gradient descent (aka back-propagation).

Unfortunately, this method does not work very well. One problem is that the gradient becomes weaker the further we move away from the data; this is known as the “**vanishing gradient**” problem (Bengio and Frasconi 1995). A related problem is that there can be large plateaus in the error surface, which cause simple first-order gadient-based methods to get stuck (Glorot and Bengio 2010).

Consequently early attempts to learn deep neural networks proved unsuccesful. Recently there has been some progress, due to the adoption of GPUs (Ciresan et al. 2010) and second-order optimization algorithms (Martens 2010). Nevertheless, such models remain difficult to train.

Below we discuss a way to initialize the parameters using unsupervised learning; this is called **generative pre-training**. The advantage of performing unsupervised learning first is that the model is forced to model a high-dimensional response, namely the input feature vector, rather than just predicting a scalar response. This acts like a data-induced regularizer, and helps backpropagation find local minima with good generalization properties (Erhan et al. 2010; Glorot and Bengio 2010).

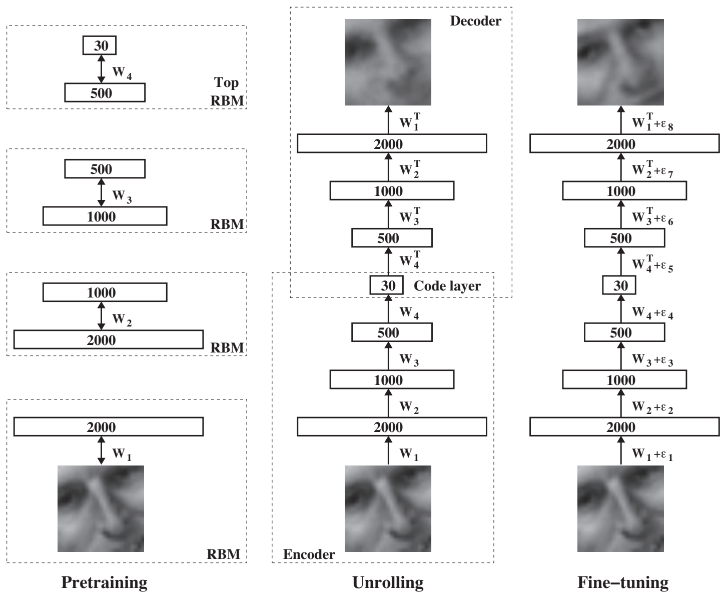


Figure 28.3 Training a deep autoencoder. (a) First we greedily train some RBMs. (b) Then we construct the auto-encoder by replicating the weights. (c) Finally we fine-tune the weights using back-propagation. From Figure 1 of (Hinton and Salakhutdinov 2006). Used with kind permission of Ruslan Salakhutdinov.

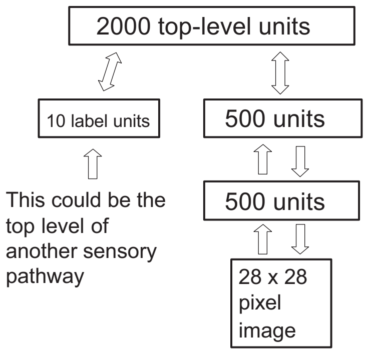
28.3.2 Deep auto-encoders

An **auto-encoder** is a kind of unsupervised neural network that is used for dimensionality reduction and feature discovery. More precisely, an auto-encoder is a feedforward neural network that is trained to predict the input itself. To prevent the system from learning the trivial identity mapping, the hidden layer in the middle is usually constrained to be a narrow **bottleneck**. The system can minimize the reconstruction error by ensuring the hidden units capture the most relevant aspects of the data.

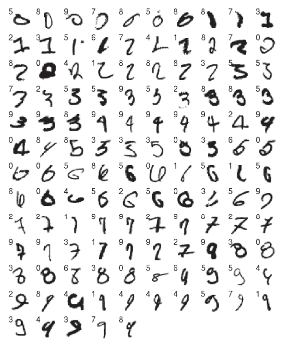
Suppose the system has one hidden layer, so the model has the form $\mathbf{v} \to \mathbf{h} \to \mathbf{v}$. Further, suppose all the functions are linear. In this case, one can show that the weights to the K hidden units will span the same subspace as the first K principal components of the data (Karhunen and Joutsensalo 1995; Japkowicz et al. 2000). In other words, linear auto-encoders are equivalent to PCA. However, by using nonlinear activation functions, one can discover nonlinear representations of the data.

More powerful representations can be learned by using **deep auto-encoders**. Unfortunately training such models using back-propagation does not work well, because the gradient signal becomes too small as it passes back through multiple layers, and the learning algorithm often gets stuck in poor local minima.

One solution to this problem is to greedily train a series of RBMs and to use these to initialize an auto-encoder, as illustrated in Figure 28.3. The whole system can then be fine-tuned using backprop in the usual fashion. This approach, first suggested in (Hinton and Salakhutdinov



(a)



(b)

Figure 28.4 (a) A DBN architecture for classifying MNIST digits. Source: Figure 1 of (Hinton et al. 2006). Used with kind permission of Geoff Hinton. (b) These are the 125 errors made by the DBN on the 10,000 test cases of MNIST. Above each image is the estimated label. Source: Figure 6 of (Hinton et al. 2006). Used with kind permission of Geoff Hinton. Compare to Figure 16.15.

2006), works much better than trying to fit the deep auto-encoder directly starting with random weights.

28.3.3 Stacked denoising auto-encoders

A standard way to train an auto-encoder is to ensure that the hidden layer is narrower than the visible layer. This prevents the model from learning the identity function. But there are other ways to prevent this trivial solution, which allow for the use of an over-complete representation. One approach is to impose sparsity constraints on the activation of the hidden units (Ranzato et al. 2006). Another approach is to add noise to the inputs; this is called a **denoising auto-encoder** (Vincent et al. 2010). For example, we can corrupt some of the inputs, for example by setting them to zero, so the model has to learn to predict the missing entries. This can be shown to be equivalent to a certain approximate form of maximum likelihood training (known as score matching) applied to an RBM (Vincent 2011).

Of course, we can stack these models on top of each other to learn a deep stacked denoising auto-encoder, which can be discriminatively fine-tuned just like a feedforward neural network, if desired.

28.4 Applications of deep networks

In this section, we mention a few applications of the models we have been discussing.

28.4.1 Handwritten digit classification using DBNs

Figure 28.4(a) shows a DBN (from (Hinton et al. 2006)) consisting of 3 hidden layers. The visible layer corresponds to binary images of handwritten digits from the MNIST data set. In addition, the top RBM is connected to a softmax layer with 10 units, representing the class label.

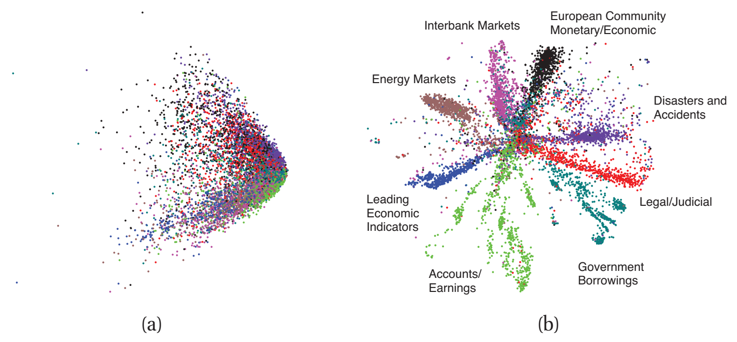


Figure 28.5 2d visualization of some bag of words data from the Reuters RCV1-v2 corpus. (a) Results of using LSA. (b) results of using a deep auto-encoder. Source: Figure 4 of (Hinton and Salakhutdinov 2006). Used with kind permission of Ruslan Salakhutdinov.

The first 2 hidden layers were trained in a greedy unsupervised fashion from 50,000 MNIST digits, using 30 epochs (passes over the data) and stochastic gradient descent, with the CD heuristic. This process took "a few hours per layer" (Hinton et al. 2006, p1540). Then the top layer was trained using as input the activations of the lower hidden layer, as well as the class labels. The corresponding generative model had a test error of about 2.5%. The network weights were then carefully fine-tuned on all 60,000 training images using the up-down procedure. This process took "about a week" (Hinton et al. 2006, p1540). The model can be used to classify by performing a deterministic bottom-up pass, and then computing the free energy for the top-level RBM for each possible class label. The final error on the test set was about 1.25%. The misclassified examples are shown in Figure 28.4(b).

This was the best error rate of any method on the permutation-invariant version of MNIST at that time. (By permutation-invariant, we mean a method that does not exploit the fact that the input is an image. Generic methods work just as well on permuted versions of the input (see Figure 1.5), and can therefore be applied to other kinds of datasets.) The only other method that comes close is an SVM with a degree 9 polynomial kernel, which has achieved an error rate of 1.4% (Decoste and Schoelkopf 2002). By way of comparison, 1-nearest neighbor (using all 60,000 examples) achieves 3.1% (see `mnist1NNdemo`). This is not as good, although 1-NN is much simpler.[3]

28.4.2 Data visualization and feature discovery using deep auto-encoders

Deep autoencoders can learn informative features from raw data. Such features are often used as input to standard supervised learning methods.

To illustrate this, consider fitting a deep auto-encoder with a 2d hidden bottleneck to some

3. One can get much improved performance on this task by exploiting the fact that the input is an image. One way to do this is to create distorted versions of the input, adding small shifts and translations (see Figure 16.13 for some examples). Applying this trick reduced the SVM error rate to 0.56%. Similar error rates can be achieved using convolutional neural networks (Section 16.5.1) trained on distorted images ((Simard et al. 2003) got 0.4%). However, the point of DBNs is that they offer a way to learn such prior knowledge, without it having to be hand-crafted.

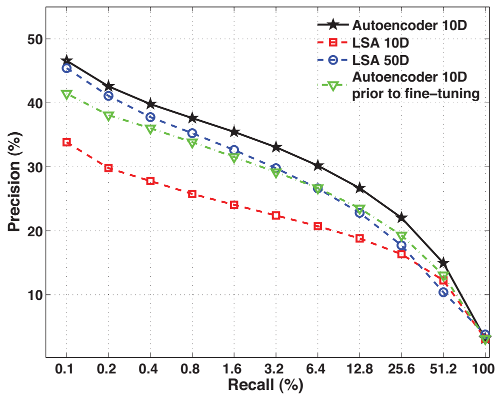


Figure 28.6 Precision-recall curves for document retrieval in the Reuters RCV1-v2 corpus. Source: Figure 3.9 of (Salakhutdinov 2009). Used with kind permission of Ruslan Salakhutdinov.

text data. The results are shown in Figure 28.5. On the left we show the 2d embedding produced by LSA (Section 27.2.2), and on the right, the 2d embedding produced by the auto-encoder. It is clear that the low-dimensional representation created by the auto-encoder has captured a lot of the meaning of the documents, even though class labels were not used.[4]

Note that various other ways of learning low-dimensional continuous embeddings of words have been proposed. See e.g., (Turian et al. 2010) for details.

28.4.3 Information retrieval using deep auto-encoders (semantic hashing)

In view of the sucess of RBMs for information retrieval discussed in Section 27.7.3.1, it is natural to wonder if deep models can do even better. In fact they can, as is shown in Figure 28.6.

More interestingly, we can use a binary low-dimensional representation in the middle layer of the deep auto-encoder, rather than a continuous representation as we used above. This enables very fast retrieval of related documents. For example, if we use a 20-bit code, we can precompute the binary representation for all the documents, and then create a hash-table mapping codewords to documents. This approach is known as **semantic hashing**, since the binary representation of semantically similar documents will be close in Hamming distance.

For the 402,207 test documents in Reuters RCV1-v2, this results in about 0.4 documents per entry in the table. At test time, we compute the codeword for the query, and then simply retrieve the relevant documents in *constant time* by looking up the contents of the relevant address in memory. To find other other related documents, we can compute all the codewords within a

4. Some details. Salakhutdinov and Hinton used the Reuters RCV1-v2 data set, which consists of 804,414 newswire articles, manually classified into 103 topics. They represent each document by counting how many times each of the top 2000 most frequent words occurs. They trained a deep auto-encoder with 2000-500-250-125-2 layers on half of the data. The 2000 visible units use a replicated softmax distribution, the 2 hidden units in the middle layer have a Gaussian distribution, and the remaining units have the usual Bernoulli-logistic distribution. When fine tuning the auto-encoder, a cross-entropy loss function (equivalent to maximum likelihood under a multinoulli distribution) was used. See (Hinton and Salakhutdinov 2006) for further details.

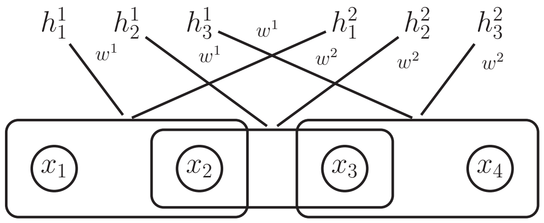


Figure 28.7 A small 1d convolutional RBM with two groups of hidden units, each associated with a filter of size 2. h^1_1 and h^2_1 are two different "views" of the data in the first window, (x_1, x_2). The first view is computed using the filter $\mathbf{w}^1$, the second view using filter $\mathbf{w}^2$. Similarly, h^1_2 and h^2_2 are the views of the data in the second window, (x_2, x_3), computed using $\mathbf{w}^1$ and $\mathbf{w}^2$ respectively.

Hamming distance of, say, 4. This results in retrieving about $6196 \times 0.4 \approx 2500$ documents[5]. The key point is that the total time is independent of the size of the corpus.

Of course, there are other techniques for fast document retrieval, such as inverted indices. These rely on the fact that individual words are quite informative, so we can simply intersect all the documents that contain each word. However, when performing image retrieval, it is clear that we do not want to work at the pixel level. Recently (Krizhevsky and Hinton 2010) showed that a deep autoencoder could learn a good semantic hashing function that outperformed previous techniques (Torralba et al. 2008; Weiss et al. 2008) on the 80 million tiny images dataset. It is hard to apply inverted indexing techniques to real-valued data (although one could imagine vector quantizing image patches).

28.4.4 Learning audio features using 1d convolutional DBNs

To apply DBNs to time series of unbounded length, it is necessary to use some form of parameter tying. One way to do this is to use **convolutional DBNs** (Lee et al. 2009; Desjardins and Bengio 2008), which use convolutional RBMs as their basic unit. These models are a generative version of convolutional neural nets discussed in Section 16.5.1. The basic idea is illustrated in Figure 28.7. The hidden activation vector for each group is computed by convolving the input vector with that group's filter (weight vector or matrix). In other words, each node within a hidden group is a weighted combination of a subset of the inputs. We compute the activaton of all the hidden nodes by "sliding" this weight vector over the input. This allows us to model **translation invariance**, since we use the same weights no matter where in the input vector the pattern occurs.[6] Each group has its own filter, corresponding to its own pattern detector.

5. Note that $6196 = \sum_{k=0}^{4} \binom{20}{k}$ is the number of bit vectors that are up to a Hamming distance of 4 away.
6. It is often said that the goal of deep learnng is to discover **invariant features**, e.g., a representation of an object that does not change even as nuisance variables, such as the lighting, do change. However, sometimes these so-called "nuisance variables" may be the variables of interest. For example if the task is to determine if a photograph was taken in the morning or the evening, then lighting is one of the more salient features, and object identity may be less relevant. As always, one task's "signal" is another task's "noise", so it unwise to "throw away" apparently irrelevant information

More formally, for binary 1d signals, we can define the full conditionals in a convolutional RBM as follows (Lee et al. 2009):

$$p(h_t^k = 1|\mathbf{v}) = \text{sigm}((\mathbf{w}^k \otimes \mathbf{v})_t + b_t) \tag{28.6}$$

$$p(v_s = 1|\mathbf{h}) = \text{sigm}(\sum_k (\mathbf{w}^k \otimes \mathbf{h}^k)_s + c_s) \tag{28.7}$$

where $\mathbf{w}^k$ is the weight vector for group k, b_t and c_s are bias terms, and $\mathbf{a} \otimes \mathbf{b}$ represents the convolution of vectors $\mathbf{a}$ and $\mathbf{b}$.

It is common to add a **max pooling** layer as well as a convolutional layer, which computes a local maximum over the filtered response. This allows for a small amount of translation invariance. It also reduces the size of the higher levels, which speeds up computation considerably. Defining this for a neural network is simple, but defining this in a way which allows for information flow backwards as well as forwards is a bit more involved. The basic idea is similar to a noisy-OR CPD (Section 10.2.3), where we define a probabilistic relationship between the max node and the parts it is maxing over. See (Lee et al. 2009) for details. Note, however, that the top-down generative process will be difficult, since the max pooling operation throws away so much information.

(Lee et al. 2009) applies 1d convolutional DBNs of depth 2 to auditory data. When the input consists of speech signals, the method recovers a representation that is similar to phonemes. When applied to music classification and speaker identification, their method outperforms techniques using standard features such as MFCC. (All features were fed into the same discriminative classifier.)

In (Seide et al. 2011), a deep neural net was used in place of a GMM inside a conventional HMM. The use of DNNs significantly improved performance on conversational speech recognition. In an interview, the tech lead of this project said "historically, there have been very few individual technologies in speech recognition that have led to improvements of this magnitude".[7]

28.4.5 Learning image features using 2d convolutional DBNs

We can extend a convolutional DBN from 1d to 2d in a straightforward way (Lee et al. 2009), as illustrated in Figure 28.8. The results of a 3 layer system trained on four classes of visual objects (cars, motorbikes, faces and airplanes) from the Caltech 101 dataset are shown in Figure 28.9. We only show the results for layers 2 and 3, because layer 1 learns Gabor-like filters that are very similar to those learned by sparse coding, shown in Figure 13.21(b). We see that layer 2 has learned some generic visual parts that are shared amongst object classes, and layer 3 seems to have learned filters that look like grandmother cells, that are specific to individual object classes, and in some cases, to individual objects.

28.5 Discussion

So far, we have been discussing models inspired by low-level processing in the brain. These models have produced useful features for simple classification tasks. But can this pure bottom-up

too early.

7. Source: `http://research.microsoft.com/en-us/news/features/speechrecognition-082911.aspx`.

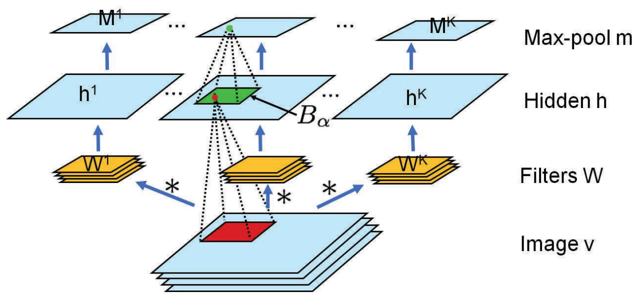


Figure 28.8 A 2d convolutional RBM with max-pooling layers. The input signal is a stack of 2d images (e.g., color planes). Each input layer is passed through a different set of filters. Each hidden unit is obtained by convolving with the appropriate filter, and then summing over the input planes. The final layer is obtained by computing the local maximum within a small window. Source: Figure 1 of (Chen et al. 2010) . Used with kind permission of Bo Chen.

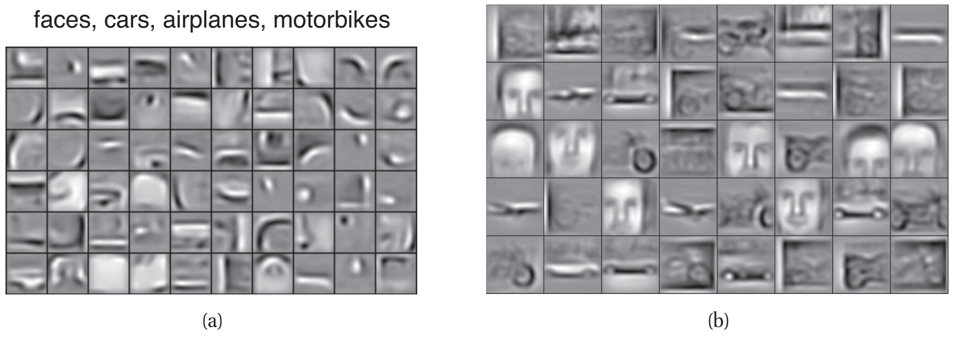


Figure 28.9 Visualization of the filters learned by a convolutional DBN in layers two and three. Source: Figure 3 of (Lee et al. 2009). Used with kind permission of Honglak Lee.

approach scale to more challenging problems, such as scene interpretation or natural language understanding?

To put the problem in perspective, consider the DBN for handwritten digit classification in Figure 28.4(a). This has about 1.6M free parameters ($28 \times 28 \times 500 + 500 \times 500 + 510 \times 2000 = 1,662,000$). Although this is a lot, it is tiny compared to the number of neurons in the brain. As Hinton says,

> This is about as many parameters as 0.002 cubic millimetres of mouse cortex, and several hundred networks of this complexity could fit within a single voxel of a high-resolution fMRI scan. This suggests that much bigger networks may be required to compete with human shape recognition abilities. — (Hinton et al. 2006, p1547).

To scale up to more challenging problems, various groups are using GPUs (see e.g., (Raina et al. 2009)) and/or parallel computing. But perhaps a more efficient approach is to work at a higher level of abstraction, where inference is done in the space of objects or their parts, rather than in the space of bits and pixels. That is, we want to bridge the **signal-to-symbol** divide, where by "symbol" we mean something atomic, that can be combined with other symbols in a compositional way.

The question of how to convert low level signals into a more structured/ "semantic" representation is known as the **symbol grounding** problem (Harnard 1990). Traditionally such symbols are associated with words in natural language, but it seems unlikely we can jump directly from low-level signals to high-level semantic concepts. Instead, what we need is an intermediate level of symbolic or atomic parts.

A very simple way to create such parts from real-valued signals, such as images, is to apply vector quantization. This generates a set of **visual words**. These can then be modelled using some of the techniques from Chapter 27 for modeling bags of words. Such models, however, are still quite "shallow".

It is possible to define, and learn, deep models which use discrete latent parts. Here we just mention a few recent approaches, to give a flavor of the possibilites. (Salakhutdinov et al. 2011) combine RBMs with hierarchical latent Dirichlet allocation methods, trained in an unsupervised way. (Zhu et al. 2010) use latent and-or graphs, trained in a manner similar to a latent structural SVM. A similar approach, based on grammars, is described in (Girshick et al. 2011). What is interesting about these techniques is that they apply data-driven machine learning methods to rich structured/symbolic "AI-style" models. This seems like a promising future direction for machine learning.