Chapter 6

Restricted Boltzmann Machines

"Available energy is the main object at stake in the struggle for existence and the evolution of the world."—Ludwig Boltzmann

6.1 Introduction

The restricted Boltzmann machine (RBM) is a fundamentally different model from the feed-forward network. Conventional neural networks are input-output mapping networks where a set of inputs is mapped to a set of outputs. On the other hand, RBMs are networks in which the probabilistic states of a network are learned for a set of inputs, which is useful for *unsupervised* modeling. While a feed-forward network minimizes a loss function of a prediction (computed from *observed* inputs) with respect to an *observed* output, a restricted Boltzmann machine models the joint probability distribution of the observed attributes together with some hidden attributes. Whereas traditional feed-forward networks have *directed* edges corresponding to the flow of computation from input to output, RBMs are *undirected* networks because they are designed to learn probabilistic *relationships* rather than input-output mappings. Restricted Boltzmann machines are probabilistic models that create latent representations of the underlying data points. Although an autoencoder can also be used to construct latent representations, a Boltzmann machine creates a *stochastic* hidden representation of each point. Most autoencoders (except for the variational autoencoder) create *deterministic* hidden representations of the data points. As a result, the RBM requires a fundamentally different way of training and using it.

C. C. Aggarwal, *Neural Networks and Deep Learning*,
https://doi.org/10.1007/978-3-319-94463-0_6

At their core, RBMs are unsupervised models that generate latent feature representations of the data points; however, the learned representations can be combined with traditional backpropagation in a closely related feed-forward network (to the specific RBM at hand) for supervised applications. This type of combination of unsupervised and supervised learning is similar to the pretraining that is performed with a traditional autoencoder architecture (cf. Section 4.7 of Chapter 4). In fact, RBMs are credited for the popularization of pretraining in the early years. The idea was soon adapted to autoencoders, which are simpler to train because of their deterministic hidden states.

6.1.1 Historical Perspective

Restricted Boltzmann machines have evolved from a classical model in the neural networks literature, which is referred to as the *Hopfield network*. This network contains nodes containing binary states, which represent binary attribute values in the training data. The Hopfield network creates a *deterministic* model of the relationships among the different attributes by using weighted edges between nodes. Eventually, the Hopfield network evolved into the notion of a Boltzmann machine, which uses *probabilistic* states to represent the Bernoulli distributions of the binary attributes. The Boltzmann machine contains both visible states and hidden states. The visible states model the distributions of the observed data points, whereas the hidden states model the distribution of the latent (hidden) variables. The parameters of the connections among the various states regulate their joint distribution. The goal is to learn the model parameters so that the likelihood of the model is maximized. The Boltzmann machine is a member of the family of (undirected) probabilistic graphical models. Eventually, the Boltzmann machine evolved into the *restricted* Boltzmann Machine (RBM). The main difference between the Boltzmann machine and the restricted Boltzmann machine is that the latter only allows connections between hidden units and visible units. This simplification is very useful from a practical point of view, because it allows the design of more efficient training algorithms. The RBM is a special case of the class of probabilistic graphical models known as *Markov random fields*.

In the initial years, RBMs were considered too slow to train and were therefore not very popular. However, at the turn of the century, faster algorithms were proposed for this class of models. Furthermore, they received some prominence as one of the ensemble components of the entry [414] winning the Netflix prize contest [577]. RBMs are generally used for unsupervised applications like matrix factorization, latent modeling, and dimensionality reduction, although there are many ways of extending them to the supervised case. It is noteworthy that RBMs usually work with binary states in their most natural form, although it is possible to work with other data types. Most of the discussion in this chapter will be restricted to units with binary states. The successful training of deep networks with RBMs preceded successful training experiences with conventional neural networks. In other words, it was shown how multiple RBMs could be stacked to create deep networks and train them effectively, before similar ideas were generalized to conventional networks.

Chapter Organization

This chapter is organized as follows. The next section will introduce Hopfield networks, which was the precursor to the Boltzmann family of models. The Boltzmann machine is introduced in Section 6.3. Restricted Boltzmann machines are introduced in Section 6.4. Applications of restricted Boltzmann machines are discussed in Section 6.5. The use of RBMs for generalized data types beyond binary representations is discussed in Section 6.6.

The process of stacking multiple restricted Boltzmann machines in order to create deep networks is discussed in Section 6.7. A summary is given in Section 6.8.

6.2 Hopfield Networks

Hopfield networks were proposed in 1982 [207] as a model to store memory. A Hopfield network is an undirected network, in which the d units (or neurons) are indexed by values drawn from $\{1 \ldots d\}$. Each connection is of the form (i, j), where each i and j is a neuron drawn from $\{1 \ldots d\}$. Each connection (i, j) is undirected, and is associated with a weight $w_{ij} = w_{ji}$. Although all pairs of nodes are assumed to have connections between them, setting w_{ij} to 0 has the effect of dropping the connection (i, j). The weight w_{ii} is set to 0, and therefore there are no self-loops. Each neuron i is associated with state s_i. An important assumption in the Hopfield network is that each s_i is a binary value drawn from $\{0, 1\}$, although one can use other conventions such as $\{-1, +1\}$. The ith node also has a bias b_i associated with it; large values of b_i encourage the ith state to be 1. The Hopfield network is an undirected model of symmetric relationships between attributes, and therefore the weights always satisfy $w_{ij} = w_{ji}$.

Each binary state in the Hopfield network corresponds to a dimension in the (binary) training data set. Therefore, if a d-dimensional training data set needs to be memorized, we need a Hopfield network with d units. The ith state in the network corresponds to the ith bit in a particular training example. The values of the states represent the binary attribute values from a training example. The weights in the Hopfield network are its parameters; large positive weights between pairs of states are indicative of high degree of positive correlation in state values, whereas large negative weights are indicative of high negative correlation. An example of a Hopfield network with an associated training data set is shown in Figure 6.1. In this case, the Hopfield network is fully connected, and the six visible states correspond to the six binary attributes in the training data.

The Hopfield network uses an optimization model to learn the weight parameters so that the weights can capture that positive and negative relationships among the attributes of the training data set. The objective function of a Hopfield network is also referred to as its *energy function*, which is analogous to the loss function of a traditional feed-forward neural network. The energy function of a Hopfield network is set up in such a way that minimizing this function encourages nodes pairs connected with large positive weights to have similar states, and pairs connected with large negative weights to have different states. The training phase of a Hopfield network, therefore, learns the weights of edges in order to minimize the energy when the states in the Hopfield network are fixed to the binary attribute values in the individual training points. Therefore, learning the weights of the Hopfield network implicitly builds an unsupervised *model* of the training data set. The energy E of a particular combination of states $\overline{s} = (s_1, \ldots s_d)$ of the Hopfield network can be defined as follows:

$$E = -\sum_i b_i s_i - \sum_{i,j:i<j} w_{ij} s_i s_j \tag{6.1}$$

The term $-b_i s_i$ encourages units with large biases to be on. Similarly, the term $-w_{ij} s_i s_j$ encourages s_i and s_j to be similar when $w_{ij} > 0$. In other words, positive weights will cause state "attraction" and negative weights will cause state "repulsion." For a small training data set, this type of modeling results in memorization, which enables one to retrieve training data points from similar, incomplete, or corrupted query points by exploring local minima of the energy function near these query points. In other words, by learning the weights of

a Hopfield network, one is implicitly memorizing the training examples, although there is a relatively conservative limit of the number of examples that can be memorized from a Hopfield network containing d units. This limit is also referred to as the *capacity* of the model.

6.2.1 Optimal State Configurations of a Trained Network

A trained Hopfield contains many local optima, each of which corresponds to either a memorized point from the training data, or a representative point in a dense region of the training data. Before discussing the training of the weights of the Hopfield network, we will discuss the methodology for finding the local energy minimum of a Hopfield network when the trained weights are already given. A local minimum is defined as a combination of states in which flipping any particular bit of the network does not reduce the energy further. The training process sets the weights in such a way that the instances in the training data tend to be local minima in the Hopfield network.

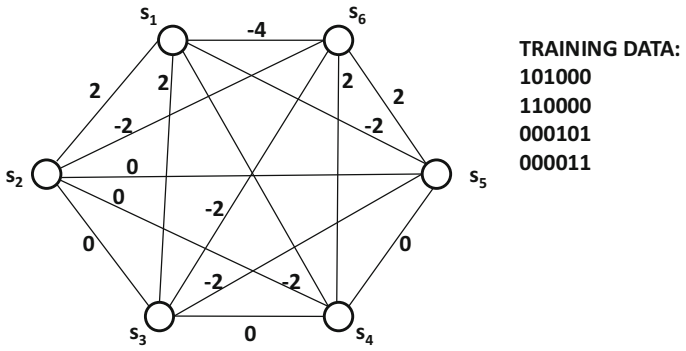


Figure 6.1: A Hopfield network with 6 visible states corresponding to 6-dimensional training data.

Finding the optimal state configuration helps the Hopfield network in recalling memories. The Hopfield network inherently learns *associative memories* because, given an input set of states (i.e., input pattern of bits), it repeatedly flips bits to improve the objective function until it finds a pattern where one cannot improve the objective function further. This local minimum (final combination of states) is often only a few bits away from the starting pattern (initial set of states), and therefore one *recalls* a closely related pattern at which a local minimum is found. Furthermore, this final pattern is often a member of the training data set (because the weights were learned using that data). In a sense, Hopfield networks provide a route towards *content-addressable memory*.

Given a starting combination of states, how can one learn the closest local minimum once the weights have already been fixed? One can use a threshold update rule to update each state in the network in order to move it towards the global energy minimum. In order to understand this point, let us compare the energy of the network between the cases when the state s_i is set to 1, and the one in which s_i is set to 0. Therefore, one can substitute two different values of s_i into Equation 6.1 to obtain the following value of the *energy gap*:

$$\Delta E_i = E_{s_i=0} - E_{s_i=1} = b_i + \sum_{j:j \neq i} w_{ij} s_j \tag{6.2}$$

This value must be larger than 0 in order for a flip of state s_i from 0 to 1 to be attractive. Therefore, one obtains the following update rule for each state s_i:

$$s_i = \begin{cases} 1 & \text{if } \sum_{j:j \neq i} w_{ij} s_j + b_i \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{6.3}$$

The above rule is iteratively used to test each state s_i and then flip the state if needed to satisfy the condition. If one is given the weights and the biases in the network at any particular time, it is possible to find a local energy minimum in terms of the states by repeatedly using the update rule above.

The local minima of a Hopfield network depend on its trained weights. Therefore, in order to "recall" a memory, one only has to provide a d-dimensional vector similar to the stored memory, and the Hopfield network will find the local minimum that is similar to this point by using it as a starting state. This type of associative memory recall is also common in humans, who often retrieve memories through a similar process of association. One can also provide a partial vector of initial states and use it to recover other states. Consider the Hopfield network shown in Figure 6.1. Note that the weights are set in such a way that each of the four training vectors in the figure will have low energy. However, there are some spurious minima such as 111000 as well. Therefore, it is not guaranteed that the local minima will always correspond to the points in the training data. However, the local minima do correspond to some key characteristics of the training data. For example, consider the spurious minimum corresponding to 111000. It is noteworthy that the first three bits are positively correlated, whereas the last three bits are also positively correlated. As a result, this minimum value of 111000 does reflect a broad pattern in the underlying data even though it is not explicitly present in the training data. It is also noteworthy that the weights of this network are closely related to the patterns in the training data. For example, the elements within the first three bits and last three bits are each positively correlated within that particular group of three bits. Furthermore, there are negative correlations *across* the two sets of elements. Consequently, the edges *within* each of the sets $\{s_1, s_2, s_3\}$ and $\{s_4, s_5, s_6\}$ tend to be positive, and those *across* these two sets are negative. Setting the weights in this data-specific way is the task of the training phase (cf. Section 6.2.2).

The iterative state update rule will arrive at one of the many local minima of the Hopfield network, depending on the initial state vector. Each of these local minima can be one of the learned "memories" from the training data set, and the closest memory to the initial state vector will be reached. These memories are implicitly stored in the weights learned during the training phase. However, it is possible for the Hopfield network to make mistakes, where closely related training patterns are merged into a single (deeper) minimum. For example, if the training data contains 1110111101 and 1110111110, the Hopfield network might learn 1110111111 as a local minimum. Therefore, in some queries, one might recover a pattern that is a small number of bits away from a pattern actually present in the training data. However, this is only a form of model generalization in which the Hopfield network is storing representative "cluster" centers instead of individual training points. In other words, the model starts generalizing instead of memorizing when the amount of data exceeds the capacity of the model; after all, Hopfield networks build unsupervised models from training data.

The Hopfield network can be used for recalling associative memories, correcting corrupted data, or for attribute completion. The tasks of recalling associative memories and cleaning corrupted data are similar. In both cases, one uses the corrupted input (or target input for associative recall) as the starting state, and uses the final state as the cleaned

output (or recalled output). In attribute completion, the state vector is initialized by setting observed states to their known values and unobserved states randomly. At this point, only the unobserved states are updated to convergence. The bit values of these states at convergence provide the completed representation.

6.2.2 Training a Hopfield Network

For a given training data set, one needs to learn the weights, so that the local minima of this network lie near instances (or dense regions) of the training data set. Hopfield networks are trained with the *Hebbian learning rule*. According to the biological motivation of Hebbian learning, a synapse between two neurons is strengthened when the neurons on either side of the synapse have highly correlated outputs. Let $x_{ij} \in \{0, 1\}$ represent the jth bit of the ith training point. The number of training instances is assumed to be n. The Hebbian learning rule sets the weights of the network as follows:

$$w_{ij} = 4\frac{\sum_{k=1}^{n}(x_{ki} - 0.5) \cdot (x_{kj} - 0.5)}{n} \tag{6.4}$$

One way of understanding this rule is that if two bits, i and j, in the training data are positively correlated, then the value $(x_{ki} - 0.5) \cdot (x_{kj} - 0.5)$ will usually be positive. As a result, the weights between the corresponding units will also be set to positive values. On the other hand, if two bits generally disagree, then the weights will be set to negative values. One can also use this rule without normalizing the denominator:

$$w_{ij} = 4\sum_{k=1}^{n}(x_{ki} - 0.5) \cdot (x_{kj} - 0.5) \tag{6.5}$$

In practice, one often wants to develop incremental learning algorithms for point-specific updates. One can update w_{ij} with only the kth training data point as follows:

$$w_{ij} \Leftarrow w_{ij} + 4(x_{ki} - 0.5) \cdot (x_{kj} - 0.5) \quad \forall i, j$$

The bias b_i can be updated by assuming that a single dummy state is always on, and the bias represents the weight between the dummy and the ith state:

$$b_i \Leftarrow b_i + 2(x_{ki} - 0.5) \quad \forall i$$

In cases where the convention is to draw the state vectors from $\{-1, +1\}$, the above rule simplifies to the following:

$$w_{ij} \Leftarrow w_{ij} + x_{ki}x_{kj} \quad \forall i, j$$
$$b_i \Leftarrow b_i + x_{ki} \quad \forall i$$

There are other learning rules, such as the *Storkey learning rule*, that are commonly used. Refer to the bibliographic notes.

Capacity of a Hopfield Network

What is the size of the training data that a Hopfield network with d visible units can store without causing errors in associative recall? It can be shown that the *storage capacity* of a Hopfield network with d units is only about $0.15 \cdot d$ training examples. Since each training

example contains d bits, it follows that the Hopfield network can store only about $0.15\,d^2$ bits. This is not an efficient form of storage because the number of weights in the network is given by $d(d-1)/2 = O(d^2)$. Furthermore, the weights are not binary and they can be shown to require $O(\log(d))$ bits. When the number of training examples is large, many errors will be made (in associative recall). These errors represent the *generalized* predictions from more data. Although it might seem that this type of generalization is useful for machine learning, there are limitations in using Hopfield networks for such applications.

6.2.3 Building a Toy Recommender and Its Limitations

Hopfield networks are often used for memorization-centric applications rather than the typical machine-learning applications requiring generalization. In order to understand the limits of a Hopfield network, we will consider an application associated with binary collaborative filtering. Since Hopfield networks work with binary data, we will assume the case of *implicit feedback* data in which user is associated with a set of binary attributes corresponding to whether or not they have watched the corresponding movies. Consider a situation in which the user Bob has watched movies *Shrek* and *Aladdin*, whereas the user Alice has watched *Gandhi*, *Nero*, and *Terminator*. It is easy to construct a fully connected Hopfield network on the universe of all movies and set the watched states to 1 and all other states to 0. This configuration can be used for each training point in order to update the weights. Of course, this approach can be extremely expensive if the base number of states (movies) is very large. For a database containing 10^6 movies, we would have 10^{12} edges, most of which will connect states containing zero values. This is because such type of implicit feedback data is often sparse, and most states will take on zero values.

One way of addressing this problem is to use *negative sampling*. In this approach, each user has their own Hopfield network containing their watched movies and a small sample of the movies that were not watched by them. For example, one might randomly sample 20 unwatched movies (of Alice) and create a Hopfield network containing $20 + 3 = 23$ states (including the watched movies). Bob's Hopfield network will containing $20 + 2 = 22$ states, and the unwatched samples might also be quite different. However, for pairs of movies that are common between the two networks, the weights will be shared. During training, all edge weights are initialized to 0. One can use repeated iterations of training over the different Hopfield networks to learn their shared weights (with the same algorithm discussed earlier). The main difference is that iterating over the different training points will lead to iterating over different Hopfield networks, each of which contains a small subset of the base network. Typically, only a small subset of the 10^{12} edges will be present in each of these networks, and most edges will never be encountered in any network. Such edges will implicitly be assumed to have weights of zero.

Now imagine a user Mary, who has watched *E.T.* and *Shrek*. We would like to recommend movies to this user. We use the full Hopfield network with only the non-zero edges present. We initialize the states for *E.T.* and *Shrek* to 1, and all other states to 0. Subsequently, we allow the updates of all states (other than *E.T.* and *Shrek*) in order to identify the minimum energy configuration of the Hopfield network. All states that are set to 1 during the updates can be recommended to the user. However, we would ideally like to have an *ordering* of the top recommended movies. One way of providing an ordering of all movies is to use the *energy gap* between the two states of each movie in order to rank the movies. The energy gap is computed only after the minimum energy configuration has been found. This approach is, however, quite naive because the final configuration of the Hopfield network is a deterministic one containing binary values, whereas the extrapolated values can only be estimated in terms of *probabilities*. For example, it would be much more natural to use

some function of the energy gap (e.g., sigmoid) in order to create probabilistic estimations. Furthermore, it would be helpful to be able to capture correlated sets of movies with some notion of latent (or hidden) states. Clearly, we need techniques in order to increase the expressive power of the Hopfield network.

6.2.4 Increasing the Expressive Power of the Hopfield Network

Although it is not standard practice, one can add *hidden units* to a Hopfield network to increase its expressive power. The hidden states serve the purpose of capturing the latent structure of the data. The weights of connections between hidden and visible units will capture the relationship between the latent structure and the training data. In some cases, it is possible to approximately represent the data only in terms of a small number of hidden states. For example, if the data contains two tightly knit clusters, one can capture this setting in two hidden states. Consider the case in which we enhance the Hopfield network of Figure 6.1 and add two hidden units. The resulting network is shown in Figure 6.2. The edges with near-zero weights have been dropped from the figure for clarity. Even though the original data is defined in terms of six bits, the two hidden units provide a *hidden* representation of the data in terms of two bits. This hidden representation is a compressed version of the data, which tells us something about the pattern at hand. In essence, all patterns are compressed to the pattern 10 or 01, depending on whether the first three bits or the last three bits dominate the training pattern. If one fixes the hidden states of the Hopfield network to 10 and randomly initializes the visible states, then one would often obtain the pattern 111000 on repeatedly using the state-wise update rule of Equation 6.3. One also obtains the pattern 0001111 as the final resting point when one starts with the hidden state 01. Notably, the patterns 000111 and 111000 are close approximations of the two types of patterns in the data, which is what one would expect from a compression technique. If we provide an incomplete version of the visible units, and then iteratively update the other states with the update rule of Equation 6.3, one would often arrive at either 000111 and 111000 depending on how the bits in the incomplete representation are distributed. If we add hidden units to a Hopfield network *and* allow the states to be probabilistic (rather than deterministic), we obtain a Boltzmann machine. This is the reason that Boltzmann machines can be viewed as *stochastic Hopfield networks with hidden units*.

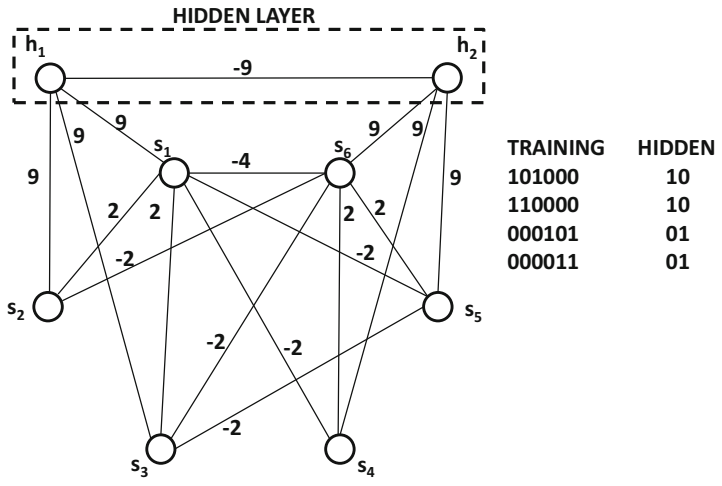


Figure 6.2: The Hopfield network with two hidden nodes

6.3 The Boltzmann Machine

Throughout this section, we assume that the Boltzmann machine contains a total of $q = (m+d)$ states, where d is the number of visible states and m is the number of hidden states. A particular state configuration is defined by the value of the state vector $\overline{s} = (s_1 \ldots s_q)$. If one explicitly wants to demarcate the visible and hidden states in $\overline{s}$, then the state vector $\overline{s}$ can be written as the pair $(\overline{v}, \overline{h})$, where $\overline{v}$ denotes the set of visible units and $\overline{h}$ denotes the set of hidden units. The states in $(\overline{v}, \overline{h})$ represent exactly the same set as $\overline{s} = \{s_1 \ldots s_q\}$, except that the visible and hidden units are explicitly demarcated in the former.

The Boltzmann machine is a probabilistic generalization of a Hopfield network. A Hopfield network *deterministically* sets each state s_i to either 1 or 0, depending on whether the energy gap ΔE_i of the state s_i is positive or negative. Recall that the energy gap of the ith unit is defined as the difference in energy between its two configurations (with other states being fixed to pre-defined values):

$$\Delta E_i = E_{s_i=0} - E_{s_i=1} = b_i + \sum_{j:j \neq i} w_{ij} s_j \tag{6.6}$$

The Hopfield network deterministically sets the value of s_i to 1, when the energy gap is positive. On the other hand, a Boltzmann machine assigns a *probability* to s_i depending on the energy gap. Positive energy gaps are assigned probabilities that are larger than 0.5. The probability of state s_i is defined by applying the sigmoid function to the energy gap:

$$P(s_i = 1 | s_1, \ldots s_{i-1}, s_{i+1}, s_q) = \frac{1}{1 + \exp(-\Delta E_i)} \tag{6.7}$$

Note that the state s_i is now a Bernoulli random variable and a zero energy gap leads to a probability of 0.5 for each binary outcome of the state.

For a particular set of parameters w_{ij} and b_i, the Boltzmann machine defines a probability distribution over various state configurations. The energy of a particular configuration $\overline{s} = (\overline{v}, \overline{h})$ is denoted by $E(\overline{s}) = E([\overline{v}, \overline{h}])$, and is defined in a similar way to the Hopfield network as follows:

$$E(\overline{s}) = -\sum_i b_i s_i - \sum_{i,j:i<j} w_{ij} s_i s_j \tag{6.8}$$

However, these configurations are only probabilistically known in the case of the Boltzmann machine (according to Equation 6.7). The conditional distribution of Equation 6.7 follows from a more fundamental definition of the unconditional probability $P(\overline{s})$ of a particular configuration $\overline{s}$:

$$P(\overline{s}) \propto \exp(-E(\overline{s})) = \frac{1}{Z} \exp(-E(\overline{s})) \tag{6.9}$$

The normalization factor Z is defined so that the probabilities over all possible configurations sum to 1:

$$Z = \sum_{\overline{s}} \exp(-E(\overline{s})) \tag{6.10}$$

The normalization factor Z is also referred to as the *partition function*. In general, the explicit computation of the partition function is hard, because it contains an exponential number of terms corresponding to all possible configurations of states. Because of the intractability of the partition function, exact computation of $P(\overline{s}) = P(\overline{v}, \overline{h})$ is not possible. Nevertheless, the computation of many types of conditional probabilities (e.g., $P(\overline{v}|\overline{h})$) is possible, because such conditional probabilities are ratios and the intractable normalization

factor gets canceled out from the computation. For example, the conditional probability of Equation 6.7 follows from the more fundamental definition of the probability of a configuration (cf. Equation 6.9) as follows:

$$P(s_i = 1|s_1, \ldots s_{i-1}, s_{i+1}, s_q) = \frac{P(s_1, \ldots s_{i-1}, \overbrace{1}^{s_i}, s_{i+1}, s_q)}{P(s_1, \ldots s_{i-1}, \underbrace{1}_{s_i}, s_{i+1}, s_q) + P(s_1, \ldots s_{i-1}, \underbrace{0}_{s_i}, s_{i+1}, s_q)}$$
$$= \frac{\exp(-E_{s_i=1})}{\exp(-E_{s_i=1}) + \exp(-E_{s_i=0})} = \frac{1}{1 + \exp(E_{s_i=1} - E_{s_i=0})}$$
$$= \frac{1}{1 + \exp(-\Delta E_i)} = \text{Sigmoid}(\Delta E_i)$$

This is the same condition as Equation 6.9. One can also see that the logistic sigmoid function finds its roots in notions of energy from statistical physics.

One way of thinking about the benefit of setting these states probabilistically is that we can now sample from these states to create new data points that look like the original data. This makes Boltzmann machines probabilistic models rather than deterministic ones. Many generative models in machine learning (e.g., Gaussian mixture models for clustering) use a sequential process of first sampling the hidden state(s) from a prior, and then generating visible observations conditionally on the hidden state(s). This is not the case in the Boltzmann machine, in which the dependence between all pairs of states is *undirected*; the visible states depend as much on the hidden states as the hidden states depend on visible states. As a result, the generation of data with a Boltzmann machine can be more challenging than in many other generative models.

6.3.1 How a Boltzmann Machine Generates Data

In a Boltzmann machine, the dynamics of the data generation is complicated by the circular dependencies among the states based on Equation 6.7. Therefore, we need an iterative process to generate sample data points from the Boltzmann machine so that Equation 6.7 is satisfied for all states. A Boltzmann machine iteratively samples the states using a conditional distribution generated from the state values in the previous iteration until *thermal equilibrium* is reached. The notion of thermal equilibrium means that we start at a random set of states, use Equation 6.7 to compute their conditional probabilities, and then sample the values of the states again using these probabilities. Note that we can iteratively generate s_i by using $P(s_i|s_1 \ldots s_{i-1}, s_{i+1}, \ldots s_q)$ in Equation 6.7. After running this process for a long time, the sampled values of the visible states provide us with random samples of generated data points. The time required to reach thermal equilibrium is referred to as the *burn-in time* of the procedure. This approach is referred to as *Gibbs sampling* or *Markov Chain Monte Carlo (MCMC) sampling*.

At thermal equilibrium, the generated points will represent the model captured by the Boltzmann machine. Note that the dimensions in the generated data points will be correlated with one another depending on the weights between various states. States with large weights between them will tend to be heavily correlated. For example, in a text-mining application in which the states correspond to the presence of words, there will be correlations among words belonging to a topic. Therefore, if a Boltzmann machine has been trained properly on a text data set, it will generate vectors containing these types of word correlations at thermal equilibrium, even when the states are randomly initialized. It is noticeable that

even generating a set of data points with the Boltzmann machine is a more complicated process compared to many other probabilistic models. For example, generating data points from a Gaussian mixture model only requires to sample points directly from the probability distribution of a sampled mixture component. On the other hand, the undirected nature of the Boltzmann machine forces us to run the process to thermal equilibrium just to generate samples. It is, therefore, an even more difficult to task to learn the weights between states for a given training data set.

6.3.2 Learning the Weights of a Boltzmann Machine

In a Boltzmann machine, we want to learn the weights in such a way so as to maximize the log-likelihood of the specific training data set at hand. The log-likelihoods of individual states are computed by using the logarithm of the probabilities in Equation 6.9. Therefore, by taking the logarithm of Equation 6.9, we obtain the following:

$$\log[P(\overline{s})] = -E(\overline{s}) - \log(Z) \tag{6.11}$$

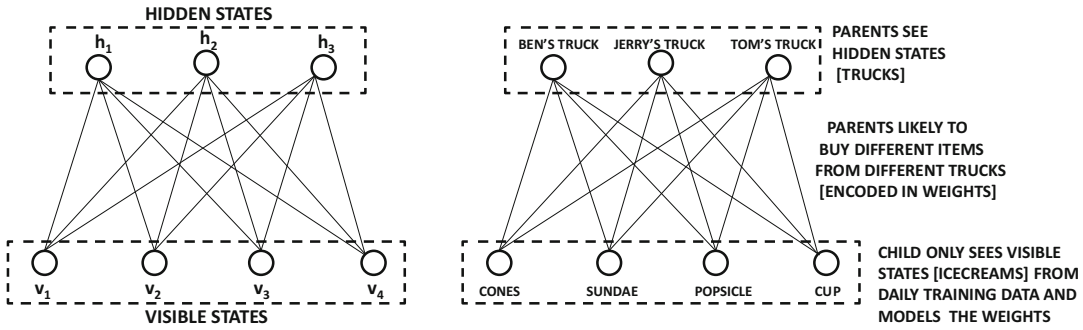
Therefore, computing $\frac{\partial \log[P(\overline{s})]}{\partial w_{ij}}$ requires the computation of the negative derivative of the energy, although we have an additional term involving the partition function. The energy function of Equation 6.8 is linear in the weight w_{ij} with coefficient of $-s_i s_j$. Therefore, the partial derivative of the energy with respect to the weight w_{ij} is $-s_i s_j$. As a result, one can show the following:

$$\frac{\partial \log[P(\overline{s})]}{\partial w_{ij}} = \langle s_i, s_j \rangle_{data} - \langle s_i, s_j \rangle_{model} \tag{6.12}$$

Here, $\langle s_i, s_j \rangle_{data}$ represents the averaged value of $s_i s_j$ obtained by running the generative process of Section 6.3.1, when the visible states are clamped to attribute values in a training point. The averaging is done over a mini-batch of training points. Similarly, $\langle s_i, s_j \rangle_{model}$ represents the averaged value of $s_i s_j$ at thermal equilibrium without fixing visible states to training points and simply running the generative process of Section 6.3.1. In this case, the averaging is done over multiple instances of running the process to thermal equilibrium. Intuitively, we want to strengthen the weights of edges between states, which tend to be *differentially* turned on together (compared to the unrestricted model), when the visible states are fixed to the training data points. This is precisely what is achieved by the update above, which uses the data- and model-centric difference in the value of $\langle s_i, s_j \rangle$. From the above discussion, it is clear that two types of samples need to be generated in order to perform the updates:

1. **Data-centric samples:** The first type of sample fixes the visible states to a randomly chosen vector from the training data set. The hidden states are initialized to random values drawn from Bernoulli distribution with probability 0.5. Then the probability of each hidden state is recomputed according to Equation 6.7. Samples of the hidden states are regenerated from these probabilities. This process is repeated for a while, so that thermal equilibrium is reached. The values of the hidden variables at this point provide the required samples. Note that the visible states are clamped to the corresponding attributes of the relevant training data vector, and therefore they do not need to be sampled.

2. **Model samples:** The second type of sample does not put any constraints on states, and one simply wants samples from the unrestricted model. The approach is the same as discussed above, except that both the visible and hidden states are initialized to random values, and updates are continuously performed until thermal equilibrium is reached.



(a) A Boltzmann machine (b) Alice's ice-cream trucks

Figure 6.3: A Restricted Boltzmann machine. Note the *restriction* of there being no interactions among either visible or hidden units.

These samples help us create an update rule for the weights. From the first type of sample, one can compute $\langle s_i, s_j \rangle_{data}$, which represents the correlations between the states of nodes i and j, when the visible vectors are fixed to a vector in the training data $\mathcal{D}$ and the hidden states are allowed to vary. Since a mini-batch of training vectors is used, one obtains multiple samples of the state vectors. The value of $\langle s_i, s_j \rangle$ is computed as the average product over all such state vectors that are obtained from Gibbs sampling. Similarly, one can estimate the value of $\langle s_i, s_j \rangle_{model}$ using the average product of s_i and s_j from the model-centric samples obtained from Gibbs sampling. Once these values have been computed, the following update is used:

$$w_{ij} \Leftarrow w_{ij} + \alpha \underbrace{(\langle s_i, s_j \rangle_{data} - \langle s_i, s_j \rangle_{model})}_{\text{Partial derivative of log probability}} \tag{6.13}$$

The update rule for the bias is similar, except that the state s_j is set to 1. One can achieve this by using a dummy bias unit that is visible and is connected to all states:

$$b_i \Leftarrow b_i + \alpha (\langle s_i, 1 \rangle_{data} - \langle s_i, 1 \rangle_{model}) \tag{6.14}$$

Note that the value of $\langle s_i, 1 \rangle$ is simply the average of the sampled values of s_i for a mini-batch of training examples from either the data-centric samples or the model-centric samples.

This approach is similar to the Hebbian update rule of a Hopfield net, except that we are also removing the effect of model-centric correlations in the update. The removal of model-centric correlations is required to account for the effect of the partition function within the expression of the log probability in Equation 6.11. The main problem with the aforementioned update rule is that it is slow in practice. This is because of the Monte Carlo sampling procedure, which requires a large number of samples in order to reach thermal equilibrium. There are faster approximations to this tedious process. In the next section, we will discuss this approach in the context of a simplified version of the Boltzmann machine, which is referred to as the *restricted* Boltzmann machine.

6.4 Restricted Boltzmann Machines

In the Boltzmann machine, the connections among hidden and visible units can be arbitrary. For example, two hidden states might contain edges between them, and so might two visible states. This type of generalized assumption creates unnecessary complexity. A natural special case of the Boltzmann machine is the *restricted* Boltzmann machine (RBM), which is bipartite, and the connections are allowed only between hidden and visible units. An example of a restricted Boltzmann machine is shown in Figure 6.3(a). In this particular example, there are three hidden nodes and four visible nodes. Each hidden state is connected to one or more visible states, although there are no connections between pairs of hidden states, and between pairs of visible states. The restricted Boltzmann machine is also referred to as a *harmonium* [457].

We assume that the hidden units are $h_1 \ldots h_m$ and the visible units are $v_1 \ldots v_d$. The bias associated with the visible node v_i be denoted by $b_i^{(v)}$, and the bias associated with hidden node h_j is denoted by $b_j^{(h)}$. Note the superscripts in order to distinguish between the biases of visible and hidden nodes. The weight of the edge between visible node v_i and hidden node h_j is denoted by w_{ij}. The notations for the weights are also slightly different for the restricted Boltzmann machine (compared to the Boltzmann machine) because the hidden and visible units are indexed separately. For example, we no longer have $w_{ij} = w_{ji}$ because the first index i always belongs to a visible node and the second index j belongs to a hidden node. It is important to keep these notational differences in mind while extrapolating the equations from the previous section.

In order to provide better interpretability, we will use a running example throughout this section, which we refer to as the example of "Alice's ice-cream trucks" based on the Boltzmann machine in Figure 6.3(b). Imagine a situation in which the training data corresponds to four bits representing the ice-creams received by Alice from her parents each day. These represent the visible states in our example. Therefore, Alice can collect 4-dimensional training points, as she receives (between 0 and 4) ice-creams of different types each day. However, the ice-creams are bought for Alice by her parents from one[1] or more of three trucks shown as the hidden states in the same figure. The identity of these trucks is hidden from Alice, although she knows that there are three trucks from which her parents procure the ice-creams (and more than one truck can be used to construct a single day's ice-cream set). Alice's parents are indecisive people, and their decision-making process is unusual because they change their mind about the selected ice-creams after selecting the trucks and vice versa. The likelihood of a particular ice-cream being picked depends on the trucks selected as well as the weights to these trucks. Similarly, the likelihood of a truck being selected depends on the ice-creams that one intends to buy and the same weights. Therefore, Alice's parents can keep changing their mind about selecting ice-creams after selecting trucks and about selecting trucks after selecting ice-creams (for a while) until they reach a final decision each day. As we will see, this *circular* relationship is the characteristic of undirected models, and process used by Alice's parents is similar to Gibb's sampling.

The use of the bipartite restriction greatly simplifies inference algorithms in RBMs, while retaining the application-centric power of the approach. If we know all the values of the visible units (as is common when a training data point is provided), the probabilities of the hidden units can be computed in one step without having to go through the laborious process of Gibbs sampling. For example, the probability of each hidden unit taking on the

[1]This example is tricky in terms of semantic interpretability for the case in which no trucks are selected. Even in that case, the probabilities of various ice-creams turn out to be non-zero depending on the bias. One can explain such cases by adding a dummy truck that is always selected.

value of 1 can be written directly as a logistic function of the values of visible units. In other words, we can apply Equation 6.7 to the restricted Boltzmann machine to obtain the following:

$$P(h_j = 1|\overline{v}) = \frac{1}{1 + \exp(-b_j^{(h)} - \sum_{i=1}^{d} v_i w_{ij})} \tag{6.15}$$

This result follows directly from Equation 6.7, which relates the state probabilities to the energy gap ΔE_j between $h_j = 0$ and $h_j = 1$. The value of ΔE_j is $b_j + \sum_i v_i w_{ij}$ when the visible states are observed. The main difference from an unrestricted Boltzmann machine is that the right-hand side of the above equation does not contain any (unknown) hidden variables and only the hidden variables. This relationship is also useful in creating a reduced representation of each training vector, once the weights have been learned. Specifically, for a Boltzmann machine with m hidden units, one can set the value of the jth hidden value to the probability computed in Equation 6.15. Note that such an approach provides a real-valued reduced representation of the binary data. One can also write the above equation using a sigmoid function:

$$P(h_j = 1|\overline{v}) = \text{Sigmoid}\left(b_j^{(h)} + \sum_{i=1}^{d} v_i w_{ij}\right) \tag{6.16}$$

One can also use a sample of the hidden states to generate the data points in one step. This is because the relationship between the visible units and the hidden units is similar in the undirected and bipartite architecture of the RBM. In other words, we can use Equation 6.7 to obtain the following:

$$P(v_i = 1|\overline{h}) = \frac{1}{1 + \exp(-b_i^{(v)} - \sum_{j=1}^{m} h_j w_{ij})} \tag{6.17}$$

One can also express this probability in terms of the sigmoid function:

$$P(v_i = 1|\overline{h}) = \text{Sigmoid}\left(b_i^{(v)} + \sum_{j=1}^{m} h_j w_{ij}\right) \tag{6.18}$$

One nice consequence of using the sigmoid is that it is often possible to create a closely related feed-forward network with sigmoid activation units in which the weights learned by the Boltzmann machine are leveraged in a directed computation with input-output mappings. The weights of this network are then fine-tuned with backpropagation. We will give examples of this approach in the application section.

Note that the weights encode the affinities between the visible and hidden states. A large positive weight implies that the two states are likely to be on together. For example, in Figure 6.3(b), it might be possible that the parents are more likely to buy cones and sundae from Ben's truck, whereas they are more likely to buy popsicles and cups from Tom's truck. These propensities are encoded in the weights, which regulate both visible state selection and hidden state selection in a circular way. The *circular* nature of the relationship creates challenges, because the relationship between ice-cream choice and truck choice runs both ways; it is the raison d'etre for Gibb's sampling. Although Alice might not know which trucks the ice-creams are coming from, she will notice the resulting correlations among the bits in the training data. In fact, if the weights of the RBM are known by Alice, she can use Gibb's sampling to generate 4-bit points representing "typical" examples of

ice-creams she will receive on future days. Even the weights of the model can be learned by Alice from examples, which is the essence of an unsupervised generative model. Given the fact that there are 3 hidden states (trucks) and enough examples of 4-dimensional training data points, Alice can learn the relevant weights and biases between the visible ice-creams and hidden trucks. An algorithm for doing this is discussed in the next section.

6.4.1 Training the RBM

Computation of the weights of the RBM is achieved using a similar type of learning rule as that used for Boltzmann machines. In particular, it is possible to create an efficient algorithm based on mini-batches. The weights w_{ij} are initialized to small values. For the current set of weights w_{ij}, they are updated as follows:

- *Positive phase:* The algorithm uses a mini-batch of training instances, and computes the probability of the state of each hidden unit in exactly one step using Equation 6.15. Then a single sample of the state of each hidden unit is generated from this probability. This process is repeated for each element in a mini-batch of training instances. The correlation between these different training instances of v_i and generated instances of h_j is computed; it is denoted by $\langle v_i, h_j \rangle_{pos}$. This correlation is essentially the average product between each such pair of visible and hidden units.

- *Negative phase:* In the negative phase, the algorithm starts with a mini-batch of training instances. Then, for each training instance, it goes through a phase of Gibbs sampling after starting with randomly initialized states. This is achieved by repeatedly using Equations 6.15 and 6.17 to compute the probabilities of the visible and hidden units, and using these probabilities to draw samples. The values of v_i and h_j at thermal equilibrium are used to compute $\langle v_i, h_j \rangle_{neg}$ in the same way as the positive phase.

- One can then use the same type of update as is used in Boltzmann machines:

$$w_{ij} \Leftarrow w_{ij} + \alpha \left(\langle v_i, h_j \rangle_{pos} - \langle v_i, h_j \rangle_{neg} \right)$$
$$b_i^{(v)} \Leftarrow b_i^{(v)} + \alpha \left(\langle v_i, 1 \rangle_{pos} - \langle v_i, 1 \rangle_{neg} \right)$$
$$b_j^{(h)} \Leftarrow b_j^{(h)} + \alpha \left(\langle 1, h_j \rangle_{pos} - \langle 1, h_j \rangle_{neg} \right)$$

  Here, $\alpha > 0$ denotes the learning rate. Each $\langle v_i, h_j \rangle$ is estimated by averaging the product of v_i and h_j over the mini-batch, although the values of v_i and h_j are computed in different ways in the positive and negative phases, respectively. Furthermore, $\langle v_i, 1 \rangle$ represents the average value of v_i in the mini-batch, and $\langle 1, h_j \rangle$ represents the average value of h_j in the mini-batch.

It is helpful to interpret the updates above in terms of Alice's trucks in Figure 6.3(b). When the weights of certain visible bits (e.g., cones and sundae) are highly correlated, the above updates will tend to push the weights in directions that these correlations can be explained by the weights between the trucks and the ice-creams. For example, if the cones and sundae are highly correlated but all other correlations are very weak, it can be explained by high weights between each of these two types of ice-creams and a single truck. In practice, the correlations will be far more complex, as will the patterns of the underlying weights.

An issue with the above approach is that one would need to run the Monte Carlo sampling for a while in order to obtain thermal equilibrium and generate the negative

samples. However, it turns out that it is possible to run the Monte Carlo sampling for only a short time *starting by fixing the visible states to a training data point from the mini-batch* and still obtain a good approximation of the gradient.

6.4.2 Contrastive Divergence Algorithm

The fastest variant of the contrastive divergence approach uses a *single* additional iteration of Monte Carlo sampling (over what is done in the positive phase) in order to generate the samples of the hidden and visible states. First, the hidden states are generated by fixing the visible units to a training point (which is already accomplished in the positive phase), and then the visible units are generated again (exactly once) from these hidden states using Monte Carlo sampling. The values of the visible units are used as the sampled states in lieu of the ones obtained at thermal equilibrium. The hidden units are generated again using these visible units. Thus, the main difference between the positive and negative phase is only of the number of iterations that one runs the approach starting with the same initialization of visible states to training points. In the positive phase, we use only half an iteration of simply computing the hidden states. In the negative phase, we use at least one *additional* iteration (so that visible states are recomputed from hidden states and hidden states generated again). This difference in the number of iterations is what causes the contrastive divergence between the state distributions in the two cases. The intuition is that an increased number of iterations causes the distribution to move away (i.e., diverge) from the data-conditioned states to what is proposed by the current weight vector. Therefore, the value of $(\langle v_i, h_j \rangle_{pos} - \langle v_i, h_j \rangle_{neg})$ in the update quantifies the amount of contrastive divergence. This fastest variant of the contrastive divergence algorithm is referred to as CD_1 because it uses a single (additional) iteration in order to generate the negative samples. Of course, using such an approach is only an approximation to the true gradient. One can improve the accuracy of contrastive divergence by increasing the number of additional iterations to k, in which the data is reconstructed k times. This approach is referred to as CD_k. Increased values of k lead to better gradients at the expense of speed.

In the early iterations, using CD_1 is good enough, although it might not be helpful in later phases. Therefore, a natural approach is to progressively increase the value of k, while applying CD_k in training. One can summarize this process as follows:

1. In the early phase of gradient-descent, the weights are initialized to small values. In each iteration, only one additional step of contrastive divergence is used. One step is sufficient at this point because the difference between the weights are very inexact in early iterations and only a rough direction of descent is needed. Therefore, even if CD_1 is executed, one will be able to obtain a good direction in most cases.

2. As the gradient descent nears a better solution, higher accuracy is needed. Therefore, two or three steps of contrastive divergence are used (i.e., CD_2 or CD_3). In general, one can double the number of Markov chain steps after a fixed number of gradient descent steps. Another approach advocated in [469] is to create the value of k in CD_k by 1 after every 10,000 steps. The maximum value of k used in [469] was 20.

The contrastive divergence algorithm can be extended to many other variations of the RBM. An excellent practical guide for training restricted Boltzmann machines may be found in [193]. This guide discusses several practical issues such as initialization, tuning, and updates. In the following, we provide a brief overview of some of these practical issues.

6.4.3 Practical Issues and Improvisations

There are several practical issues in training the RBM with contrastive divergence. Although we have always assumed that the Monte Carlo sampling procedure generates binary samples, this is not quite the case. Some of the iterations of the Monte Carlo sampling directly use *computed* probabilities (cf. Equations 6.15 and 6.17), rather than *sampled* binary values. This is done in order to reduce the noise in training, because probability values retain more information than binary samples. However, there are some differences between how hidden states and visible states are treated:

- *Improvisations in sampling hidden states:* The final iteration of CD_k computes hidden states as probability values according to Equation 6.15 for positive and negative samples. Therefore, the value of h_j used for computing $\langle v_i, h_j \rangle_{pos} - \langle v_i, h_j \rangle_{neg}$ would always be a real value for both positive and negative samples. This real value is a fraction because of the use of the sigmoid function in Equation 6.15.

- *Improvisations in sampling visible states:* Therefore, the improvisations for Monte Carlo sampling of visible states are always associated with the computation of $\langle v_i, h_j \rangle_{neg}$ rather than $\langle v_i, h_j \rangle_{pos}$ because visible states are always fixed to the training data. For the negative samples, the Monte Carlo procedure *always* computes probability values of visible states according to Equation 6.17 over *all* iterations rather than using 0-1 values. This is not the case for the hidden states, which are always binary until the very last iteration.

Using probability values iteratively rather than sampled binary values is technically incorrect, and does not reach correct thermal equilibrium. However, the contrastive divergence algorithm is an approximation anyway, and this type of approach reduces significant noise at the expense of some theoretical incorrectness. Noise reduction is a result of the fact that the probabilistic outputs are closer to expected values.

The weights can be initialized from a Gaussian distribution with zero mean and a standard deviation of 0.01. Large values of the initial weights can speed up the learning, but might lead to a model that is slightly worse in the end. The visible biases are initialized to $\log(p_i/(1-p_i))$, where p_i is the fraction of data points in which the ith dimension takes on the value of 1. The values of the hidden biases are initialized to 0.

The size of the mini-batch should be somewhere between 10 and 100. The order of the examples should be randomized. For cases in which class labels are associated with examples, the mini-batch should be selected in such a way that the proportion of labels in the batch is approximately the same as the whole data.

6.5 Applications of Restricted Boltzmann Machines

In this section, we will study several applications of restricted Boltzmann machines. These methods have been very successful for a variety of unsupervised applications, although they are also used for supervised applications. When using an RBM in a real-world application, a mapping from input to output is often required, whereas a vanilla RBM is only designed to learn probability distributions. The input-to-output mapping is often achieved by constructing a feed-forward network with weights derived from the learned RBM. In other words, one can often derive a traditional neural network that is *associated* with the original RBM.

Here, we will like to discuss the differences between the notions of the *state* of a node in the RBM, and the *activation* of that node in the associated neural network. The state of a

node is a binary value sampled from the Bernoulli probabilities defined by Equations 6.15 and 6.17. On the other hand, the activation of a node in the associated neural network is the probability value derived from the use of the sigmoid function in Equations 6.15 and 6.17. Many applications use the activations in the nodes of the associated neural network, rather than the states in the original RBM after the training. Note that the final step in the contrastive divergence algorithm also leverages the activations of the nodes rather than the states while updating the weights. In practical settings, the activations are more information-rich and are therefore useful. The use of activations is consistent with traditional neural network architectures, in which backpropagation can be used. The use of a final phase of backpropagation is crucial in being able to apply the approach to supervised applications. In most cases, the critical role of the RBM is to perform unsupervised feature learning. Therefore, the role of the RBM is often only one of pretraining in the case of supervised learning. In fact, pretraining is one of the important historical contributions of the RBM.

6.5.1 Dimensionality Reduction and Data Reconstruction

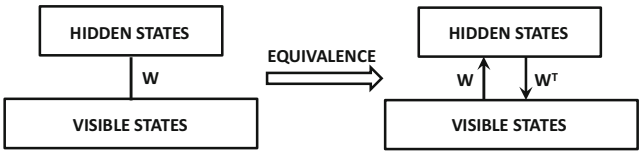
The most basic function of the RBM is that of dimensionality reduction and unsupervised feature engineering. The hidden units of an RBM contain a reduced representation of the data. However, we have not yet discussed how one can reconstruct the original representation of the data with the use of an RBM (much like an autoencoder). In order to understand the reconstruction process, we first need to understand the equivalence of the undirected RBM with directed graphical models [251], in which the computation occurs in a particular direction. Materializing a directed probabilistic graph is the first step towards materializing a traditional neural network (derived from the RBM) in which the discrete probabilistic sampling from the sigmoid can be replaced with real-valued sigmoid activations.

Although an RBM is an undirected graphical model, one can "unfold" an RBM in order to create a directed model in which the inference occurs in a particular direction. In general, an undirected RBM can be shown to be equivalent to a directed graphical model with an infinite number of layers. The unfolding is particularly useful when the visible units are fixed to specific values because the number of layers in the unfolding collapses to exactly twice the number of layers in the original RBM. Furthermore, by replacing the discrete probabilistic sampling with continuous sigmoid units, this directed model functions as a virtual autoencoder, which has both an encoder portion and a decoder portion. Although the weights of an RBM have been trained using discrete probabilistic sampling, they can also be used in this related neural network with some fine tuning. This is a heuristic approach to convert what has been learned from a Boltzmann machine (i.e., the weights) into the initialized weights of a traditional neural network with sigmoid units.

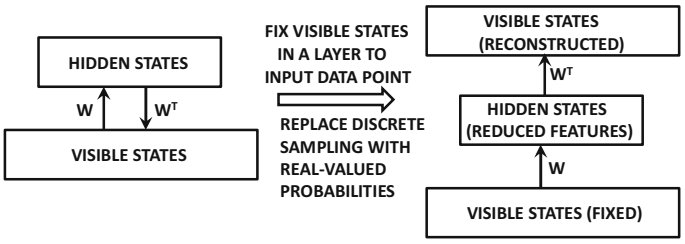
An RBM can be viewed as an undirected graphical model that uses the same weight matrix to learn $\overline{h}$ from $\overline{v}$ as it does from $\overline{v}$ to $\overline{h}$. If one carefully examines Equations 6.15 and 6.17, one can see that they are very similar. The main difference is that these equations uses different biases, and they use the transposes of each other's weight matrices. In other words, one can rewrite Equations 6.15 and 6.17 in the following form for some function $f(\cdot)$:

$$\overline{h} \sim f(\overline{v}, \overline{b}^{(h)}, W)$$
$$\overline{v} \sim f(\overline{h}, \overline{b}^{(v)}, W^T)$$

The function $f(\cdot)$ is typically defined by the sigmoid function in binary RBMs, which constitute the predominant variant of this class of models. Ignoring the biases, one can replace



(a) Equivalence of directed and undirected relationships



(b) Discrete graphical model to approximate real-valued neural network

Figure 6.4: Using trained RBM to approximate trained autoencoder

the undirected graph of the RBM with two directed links, as shown in Figure 6.4(a). Note that the weight matrices in the two directions are W and W^T, respectively. However, if we fix the visible states to the training points, we can perform just two iterations of these operations to reconstruct the visible states with *real-valued* approximations. In other words, we approximate this trained RBM with a traditional neural network by replacing discrete sampling with continuous-valued sigmoid activations (as a heuristic). This conversion is shown in Figure 6.4(b). In other words, instead of using the sampling operation of "$\sim$," we replace the samples with the probability values:

$$\overline{h} = f(\overline{v}, \overline{b}^{(h)}, W)$$
$$\overline{v}' = f(\overline{h}, \overline{b}^{(v)}, W^T)$$

Note that $\overline{v}'$ is the reconstructed version of $\overline{v}$ and it will contain real values (unlike the binary states in $\overline{v}$). In this case, we are working with real-valued activations rather than discrete samples. Because sampling is no longer used and all computations are performed in terms of expectations, we need to perform only one iteration of Equation 6.15 in order to learn the reduced representation. Furthermore, only one iteration of Equation 6.17 is required to learn the reconstructed data. The prediction phase works only in a single direction from the input point to the reconstructed data, and is shown on the right-hand side of Figure 6.4(b). We modify Equations 6.15 and 6.17 to define the states of this traditional neural network as real values:

$$\hat{h}_j = \frac{1}{1 + \exp(-b_j^{(h)} - \sum_{i=1}^{d} v_i w_{ij})} \tag{6.19}$$

For a setting with a total of $m \ll d$ hidden states, the real-valued reduced representation is given by $(\hat{h}_1 \ldots \hat{h}_m)$. This first step of creating the hidden states is equivalent to the encoder portion of an autoencoder, and these values are the expected values of the binary states. One can then apply Equation 6.17 to these *probabilistic values* (without creating Monte-Carlo instantiations) in order to reconstruct the visible states as follows:

$$\hat{v}_i = \frac{1}{1 + \exp(-b_i^{(v)} - \sum_j \hat{h}_j w_{ij})} \tag{6.20}$$

Although $\hat{h}_j$ does represent the expected value of the jth hidden unit, applying the sigmoid function again to this real-valued version of $\hat{h}_j$ only provides a rough approximation to the expected value of v_i. Nevertheless, the real-valued prediction $\hat{v}_i$ is an approximate reconstruction of v_i. Note that in order to perform this reconstruction we have used similar operations as traditional neural networks with sigmoid units rather than the troublesome discrete samples of probabilistic graphical models. Therefore, we can now use this related neural network as a good starting point for fine-tuning the weights with traditional backpropagation. This type of reconstruction is similar to the reconstruction used in the autoencoder architecture discussed in Chapter 2.

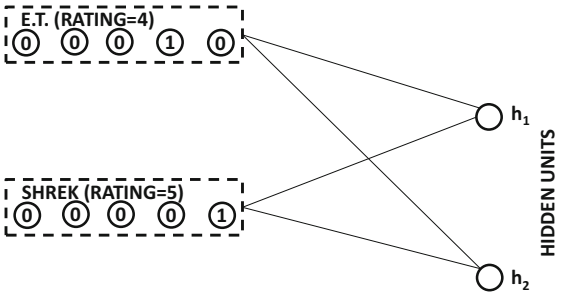
On first impression, it makes little sense to train an RBM when similar goals can be achieved with a traditional autoencoder. However, this broad approach of deriving a traditional neural network with a trained RBM is particularly useful when working with stacked RBMs (cf. Section 6.7). The training of a stacked RBM does not face the same challenges as those associated with deep neural networks, especially the ones related with the vanishing and exploding gradient problems. Just as the simple RBM provides an excellent initialization point for the shallow autoencoder, the stacked RBM also provides an excellent starting point for a deep autoencoder [198]. This principle led to the development of the idea of pretraining with RBMs before conventional pretraining methods were developed without the use of RBMs. As discussed in this section, one can also use RBMs for other reduction-centric applications such as collaborative filtering and topic modeling.

6.5.2 RBMs for Collaborative Filtering

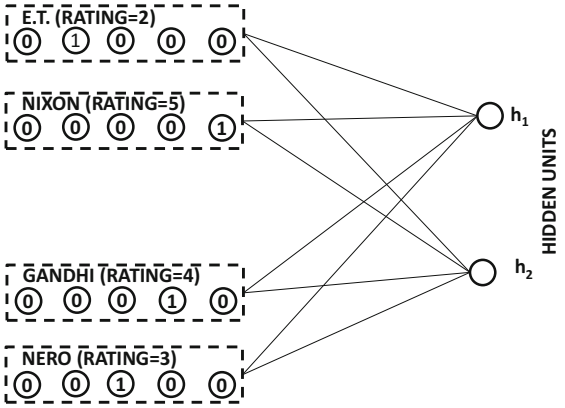
The previous section shows how restricted Boltzmann machines are used as alternatives to the autoencoder for unsupervised modeling and dimensionality reduction. However, as discussed in Section 2.5.7 of Chapter 2, dimensionality reduction methods are also used for a variety of related applications like collaborative filtering. In the following, we will provide an RBM-centric alternative to the recommendation technique described in Section 2.5.7 of Chapter 2. This approach is based on the technique proposed in [414], and it was one of the ensemble components of the winning entry in the Netflix prize contest.

One of the challenges in working with ratings matrices is that they are incompletely specified. This tends to make the design of a neural architecture for collaborative filtering more difficult than traditional dimensionality reduction. Recall from the discussion in Section 2.5.7 that modeling such incomplete matrices with a traditional neural network also faces the same challenge. In that section, it was shown how one could create a different training instance *and* a different neural network for each user, depending on which ratings are observed by that user. All these different neural networks share weights. An exactly similar approach is used with the restricted Boltzmann machine, in which one training case and one RBM is defined for each user. However, in the case of the RBM, one additional problem is that the units are binary, whereas ratings can take on values from 1 to 5. Therefore, we need some way of working with the additional constraint.

In order to address this issue, the hidden units in the RBM are allowed to be 5-way softmax units in order to correspond to rating values from 1 to 5. In other words, the hidden units are defined in the form of a one-hot encoding of the rating. One-hot encodings are naturally modeled with softmax, which defines the probabilities of each possible position. The ith softmax unit corresponds to the ith movie and the probability of a particular rating being given to that movie is defined by the distribution of softmax probabilities. Therefore, if there are d movies, we have a total of d such one-hot encoded ratings. The values of the corresponding binary values of the one-hot encoded visible units are denoted by $v_i^{(1)}, \ldots v_i^{(5)}$.



(a) RBM architecture for user Sayani (Observed Ratings: *E.T.* and *Shrek*)



(b) RBM architecture for user Bob (Observed Ratings: *E.T.*, *Nixon*, *Gandhi*, and *Nero*)

Figure 6.5: The RBM architectures of two users are shown based on their observed ratings. It is instructive to compare this figure with the conventional neural architecture shown in Figure 2.14 in Chapter 2. In both cases, weights are shared by user-specific networks.

Note that only one of the values of $v_i^{(k)}$ can be 1 over fixed i and varying k. The hidden layer is assumed to contain m units. The weight matrix has a separate parameter for each of the multinomial outcomes of the softmax unit. Therefore, the weight between visible unit i and hidden unit j for the outcome k is denoted by $w_{ij}^{(k)}$. In addition, we have 5 biases for the visible unit i, which are denoted by $b_i^{(k)}$ for $k \in \{1, \ldots, 5\}$. The hidden units only have a single bias, and the bias of the jth hidden unit is denoted by b_j (without a superscript). The architecture of the RBM for collaborative filtering is illustrated in Figure 6.5. This example contains $d = 5$ movies and $m = 2$ hidden units. In this case, the RBM architectures of two users, Sayani and Bob, are shown in the figure. In the case of Sayani, she has specified ratings for only two movies. Therefore, a total of $2 \times 2 \times 5 = 20$ connections will be present in her case, even though we have shown only a subset of them to avoid clutter in the figure. In the case of Bob, he has four observed ratings, and therefore his network will contain a total of $4 \times 2 \times 5 = 40$ connections. Note that both Sayani and Bob have rated the movie *E.T.*, and therefore the connections from this movie to the hidden units will share weights between the corresponding RBMs.

The states of the hidden units, which are binary, are defined with the use of the sigmoid function:

$$P(h_j = 1|\overline{v}^{(1)} \ldots \overline{v}^{(5)}) = \frac{1}{1 + \exp(-b_j - \sum_{i,k} v_i^{(k)} w_{ij}^k)} \tag{6.21}$$

The main difference from Equation 6.15 is that the visible units also contain a superscript to correspond to the different rating outcomes. Otherwise, the condition is virtually identical. However, the probabilities of the visible units are defined differently from the traditional RBM model. In this case, the visible units are defined using the softmax function:

$$P(v_i^{(k)} = 1|\overline{h}) = \frac{\exp(b_i^{(k)} + \sum_j h_j w_{ij}^{(k)})}{\sum_{r=1}^{5} \exp(b_i^{(r)} + \sum_j h_j w_{ij}^{(r)})} \tag{6.22}$$

The training is done in a similar way as the unrestricted Boltzmann machine with Monte Carlo sampling. The main difference is that the visible states are generated from a multinomial model. Therefore, the MCMC sampling should also generate the negative samples from the multinomial model of Equation 6.22 to create each $v_i^{(k)}$. The corresponding updates for training the weights are as follows:

$$w_{ij}^{(k)} \Leftarrow w_{ij}^{(k)} + \alpha \left(\langle v_i^{(k)}, h_j \rangle_{pos} - \langle v_i^{(k)}, h_j \rangle_{neg} \right) \quad \forall k \tag{6.23}$$

Note that only the weights of the *observed* visible units to all hidden units are updated for a single training example (i.e., user). In other words, the Boltzmann machine that is used is different for each user in the data, although the weights are shared across the different users. Examples of the Boltzmann machines for two different training examples are illustrated in Figure 6.5, and the architectures for Bob and Sayani are different. However, the weights for the units representing *E.T.* are shared. This type of approach is also used in the traditional neural architecture of Section 2.5.7 in which the neural network used for each training example is different. As discussed in that section, the traditional neural architecture is equivalent to a matrix factorization technique. The Boltzmann machine tends to give somewhat different ratings predictions from matrix factorization techniques, although the accuracy is similar.

Making Predictions

Once the weights have been learned, they can be used for making predictions. However, the predictive phase works with real-valued activations rather than binary states, much like a traditional neural network with sigmoid and softmax units. First, one can use Equation 6.21 in order to learn the probabilities of the hidden units. Let the probability that the jth hidden unit is 1 be denoted by $\hat{p}_j$. Then, the probabilities of *unobserved* visible units are computed using Equation 6.22. The main problem in computing Equation 6.22 is that it is defined in terms of the values of the hidden units, which are only known in the form of probabilities according to Equation 6.21. However, one can simply replace each h_j with $\hat{p}_j$ in Equation 6.22 in order to compute the probabilities of the visible units. Note that these predictions provide the probabilities of each possible rating value of each item. These probabilities can also be used to compute the expected value of the rating if needed. Although this approach is approximate from a theoretical point of view, it works well in practice and is extremely fast. By using these real-valued computations, one is effectively converting the RBM into a traditional neural network architecture with logistic units for hidden layers and

softmax units for the input and output layers. Although the original paper [414] does not mention it, it is even possible to tune the weights of this network with backpropagation (cf. Exercise 1).

The RBM approach works as well as the traditional matrix factorization approach, although it tends to give different types of predictions. This type of diversity is an advantage from the perspective of using an ensemble-centric approach. Therefore, the results can be combined with the matrix factorization approach in order to yield the improvements that are naturally associated with an ensemble method. Ensemble methods generally show better improvements when diverse methods of similar accuracy are combined.

Conditional Factoring: A Neat Regularization Trick

A neat regularization trick is buried inside the RBM-based collaborative filtering work of [414]. This trick is not specific to the collaborative filtering application, but can be used in any application of an RBM. This approach is not necessary in traditional neural networks, where it can be simulated by incorporating an additional hidden layer, but it is particularly useful for RBMs. Here, we describe this trick in a more general way, without its specific modifications for the collaborative filtering application. In some applications with a large number of hidden units and visible units, the size of the parameter matrix $W = [w_{ij}]$ might be large. For example, in a matrix with $d = 10^5$ visible units, and $m = 100$ hidden units, we will have ten million parameters. Therefore, more than ten million training points will be required to avoid overfitting. A natural approach is to assume a low-rank parameter structure of the weight matrix, which is a form of regularization. The idea is to assume that the matrix W can be expressed as the product of two low-rank factors U and V, which are of sizes $d \times k$ and $m \times k$, respectively. Therefore, we have the following:

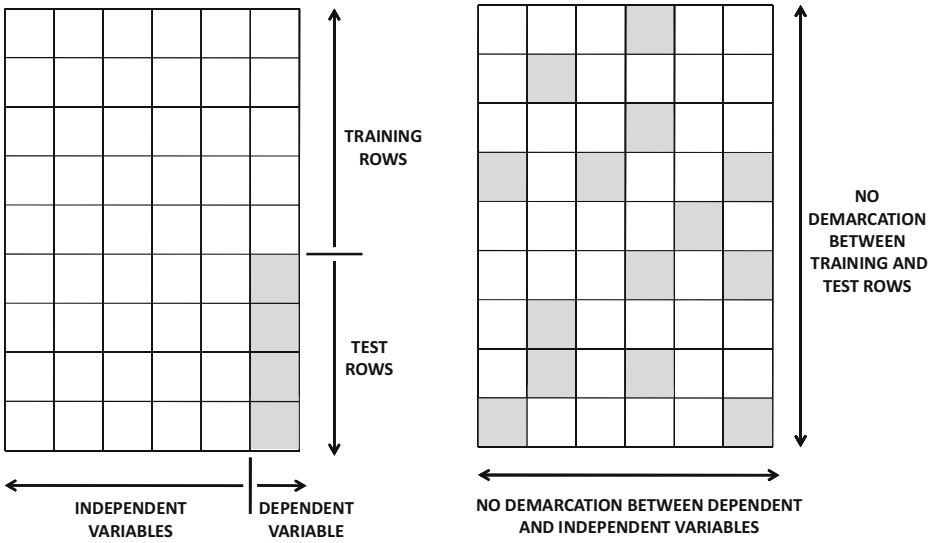
$$W = UV^T \tag{6.24}$$

Here, k is the rank of the factorization, which is typically much less than both d and m. Then, instead of learning the parameters of the matrix W, one can learn the parameters of U and V, respectively. This type of trick is used often in various machine learning applications, where parameters are represented as a matrix. A specific example is that of factorization machines, which are also used for collaborative filtering [396]. This type of approach is not required in traditional neural networks, because one can simulate it by incorporating an additional linear layer with k units between two layers with a weight matrix of W between them. The weight matrices of the two layers will be U and V^T, respectively.

6.5.3 Using RBMs for Classification

The most common way to use RBMs for classification is as a pretraining procedure. In other words, a Boltzmann machine is first used to perform unsupervised feature engineering. The RBM is then unrolled into a related encoder-decoder architecture according to the approach described in Section 6.5.1. This is a traditional neural network with sigmoid units, whose weights are derived from the unsupervised RBM rather than backpropagation. The encoder portion of this neural network is topped with an output layer for class prediction. The weights of this neural network are then fine-tuned with backpropagation. Such an approach can even be used with *stacked RBMs* (cf. Section 6.7) to yield a deep classifier. This methodology of initializing a (conventional) deep neural network with an RBM was one of the first approaches for pretraining deep networks.

There is, however, another alternative approach to perform classification with the RBM, which integrates RBM training and inference more tightly with the classification process. This approach is somewhat similar to the collaborative filtering methodology discussed in the previous section. The collaborative-filtering problem is also referred to as *matrix completion* because the missing entries of an incompletely specified matrix are predicted. The use of RBMs for recommender systems provides some useful hints about their use in classification. This is because classification can be viewed as a simplified version of the matrix completion problem in which we create a single matrix out of both the training and test rows, and the missing values belong to a particular column of the matrix. This column corresponds to the class variable. Furthermore, all the missing values are present in the test rows in the case of classification, whereas the missing values could be present anywhere in the matrix in the case of recommender systems. This relationship between classification and the generic matrix completion problem is illustrated in Figure 6.6. In classification, all features are observed for the rows corresponding to training points, which simplifies the modeling (compared to collaborative filtering in which a complete set of features is typically not observed for any row).



(a) Classification (b) Matrix completion

Figure 6.6: The classification problem is a special case of matrix completion. Shaded entries are missing and need to be predicted.

We assume that the input data contains d binary features. The class label has k discrete values, which corresponds to the multiway classification problem. The classification problem can be modeled by the RBM by defining the hidden and visible features as follows:

1. The visible layer contains two types of nodes corresponding to the features and the class label, respectively. There are d binary units corresponding to features, and there are k binary units corresponding to the class label. However, only one of these k binary units can take on the value of 1, which corresponds to a one-hot encoding of the class labels. This encoding of the class label is similar to the approach used for encoding the ratings in the collaborative-filtering application. The visible units for the features are denoted by $v_1^{(f)} \ldots v_d^{(f)}$, whereas the visible units for the class labels are denoted by $v_1^{(c)} \ldots v_k^{(c)}$. Note that the symbolic superscripts denote whether the visible units corresponds to a feature or a class label.

2. The hidden layer contains m binary units. The hidden units are denoted by $h_1 \ldots h_m$.

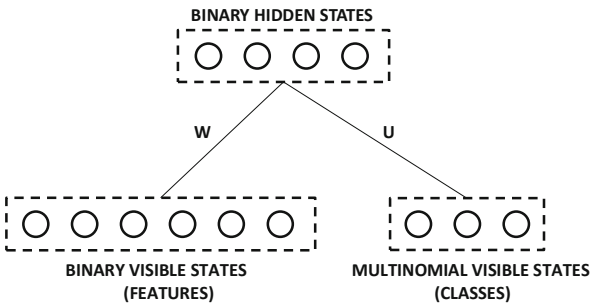


Figure 6.7: The RBM architecture for classification

The weight of the connection between the ith feature-specific visible unit $v_i^{(f)}$ and the jth hidden unit h_j is given by w_{ij}. This results in a $d \times m$ connection matrix $W = [w_{ij}]$. The weight of the connection between the ith class-specific visible unit $v_i^{(c)}$ and the jth hidden unit h_j is given by u_{ij}. This results in a $k \times m$ connection matrix $U = [u_{ij}]$. The relationships between different types of nodes and matrices for $d = 6$ features, $k = 3$ classes, and $m = 5$ hidden features is shown in Figure 6.7. The bias for the ith feature-specific visible node is denoted by $b_i^{(f)}$, and the bias for the ith class-specific visible node is denoted by $b_i^{(c)}$. The bias for the jth hidden node is denoted by b_j (with no superscript). The states of the hidden nodes are defined in terms of all visible nodes using the sigmoid function:

$$P(h_j = 1|\overline{v}^{(f)}, \overline{v}^{(c)}) = \frac{1}{1 + \exp(-b_j - \sum_{i=1}^{d} v_i^{(f)} w_{ij} - \sum_{i=1}^{k} v_i^{(c)} u_{ij})} \tag{6.25}$$

Note that this is the standard way in which the probabilities of hidden units are defined in a Boltzmann machine. There are, however, some differences between how the probabilities of the feature-specific visible units and the class-specific visible units are defined. In the case of the feature-specific visible units, the relationship is not very different from a standard Boltzmann machine:

$$P(v_i^{(f)} = 1|\overline{h}) = \frac{1}{1 + \exp(-b_i^{(f)} - \sum_{j=1}^{m} h_j w_{ij})} \tag{6.26}$$

The case of the class units is, however, slightly different because we must use the softmax function instead of the sigmoid. This is because of the one-hot encoding of the class. Therefore, we have the following:

$$P(v_i^{(c)} = 1|\overline{h}) = \frac{\exp(b_i^{(c)} + \sum_j h_j u_{ij})}{\sum_{l=1}^{k} \exp(b_l^{(c)} + \sum_j h_j u_{lj})} \tag{6.27}$$

A naive approach to training the Boltzmann machine would use a similar generative model to previous sections. The multinomial model is used to generate the visible states $v_i^{(c)}$ for the classes. The corresponding updates of the contrastive divergence algorithm are as follows:

$$w_{ij} \Leftarrow w_{ij} + \alpha \left(\langle v_i^{(f)}, h_j \rangle_{pos} - \langle v_i^{(f)}, h_j \rangle_{neg} \right) \quad \text{if } i \text{ is feature unit}$$

$$u_{ij} \Leftarrow u_{ij} + \alpha \left(\langle v_i^{(c)}, h_j \rangle_{pos} - \langle v_i^{(c)}, h_j \rangle_{neg} \right) \quad \text{if } i \text{ is class unit}$$

This approach is a direct extension from collaborative filtering. However, the main problem is that this *generative* approach does not fully optimize for classification accuracy. To provide an analogy with autoencoders, one would not necessarily perform significantly better dimensionality reduction (in a supervised sense) by simply including the class variable among the inputs. The reduction would often be dominated by the unsupervised relationships among the features. Rather, the *entire focus* of the learning should be on optimizing the accuracy of classification. Therefore, a *discriminative* approach to training the RBM is often used in which the weights are learned to maximize the conditional class likelihood of the true class. Note that it is easy to set up the conditional probability of the class variable, given the visible states by using the probabilistic dependencies between the hidden features and classes/features. For example, in the traditional form of a restrictive Boltzmann machine, we are maximizing the *joint* probability of the feature variables $v_i^{(f)}$ and the class variables v_i^c. However, in the discriminative variant, the objective function is set up to maximize the *conditional* probability of the class variable $y \in \{1 \ldots k\}$ $P(v_y^{(c)} = 1|\overline{v}^{(f)})$. Such an approach has a more focused effect of maximizing classification accuracy. Although it is possible to train a discriminative restricted Boltzmann machine using contrastive divergence, the problem is simplified because one can estimate $P(v_y^{(c)} = 1|\overline{v}^{(f)})$ in closed form without having to use an iterative approach. This form can be shown to be the following [263, 414]:

$$P(v_y^{(c)} = 1|\overline{v}^{(f)}) = \frac{\exp(b_y^{(c)}) \prod_{j=1}^{m} [1 + \exp(b_j^{(h)} + u_{yj} + \sum_i w_{ij} v_i^{(f)})]}{\sum_{l=1}^{k} \exp(b_l^{(c)}) \prod_{j=1}^{m} [1 + \exp(b_j^{(h)} + u_{lj} + \sum_i w_{ij} v_i^{(f)})]} \tag{6.28}$$

With this differentiable closed form, it is a simple matter to differentiate the negative logarithm of the above expression for stochastic gradient descent. If $\mathcal{L}$ is the negative logarithm of the above expression and θ is any particular parameter (e.g., weight or bias) of the Boltzmann machine, one can show the following:

$$\frac{\partial \mathcal{L}}{\partial \theta} = \sum_{j=1}^{m} \text{Sigmoid}(o_{yj}) \frac{\partial o_{yj}}{\partial \theta} - \sum_{l=1}^{k} \sum_{j=1}^{m} \text{Sigmoid}(o_{lj}) \frac{\partial o_{lj}}{\partial \theta} \tag{6.29}$$

Here, we have $o_{yj} = b_j^{(h)} + u_{yj} + \sum_i w_{ij} v_i^{(f)}$. The above expression can be easily computed for each training point and for each parameter in order to perform the stochastic gradient descent process. It is a relatively simple matter to make probabilistic predictions for unseen test instances using Equation 6.28. More details and extensions are discussed in [263].

6.5.4 Topic Models with RBMs

Topic modeling is a form of dimensionality reduction that is specific to text data. The earliest topic models, which correspond to Probabilistic Latent Semantic Analysis (PLSA), were proposed in [206]. In PLSA, the basis vectors are not orthogonal to one another, as is the case with SVD. On the other hand, both the basis vectors and the transformed representations are constrained to be nonnegative values. The nonnegativity in the value of each transformed feature is semantically useful, because it represents the strength of a topic in a particular document. In the context of the RBM, this strength corresponds to the probability that a particular hidden unit takes on the value of 1, given that the words in a particular document have been observed. Therefore, one can use the vector of conditional probabilities of the hidden states (when visible states are fixed to document words) in order to create a reduced representation of each document. It is assumed that the lexicon size is d, whereas the number of hidden units is $m \ll d$.

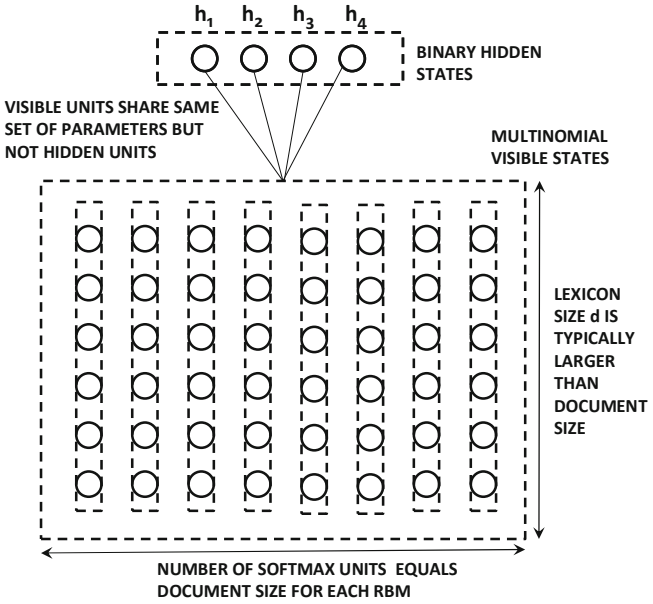


Figure 6.8: The RBM for each document is illustrated. The number of visible units is equal to the number of words in each document

This approach shares some similarities with the technique used for collaborative filtering in which a single RBM is created for each user (row of the matrix). In this case, a single RBM is created for each document. A group of visible units is created for each word, and therefore the number of groups of visible units is equal to the number of words in the document. In the following, we will concretely define how the visible and hidden states of the RBM are fixed in order to describe the concrete workings of the model:

1. For the tth document containing n_t words, a total of n_t softmax groups are retained. Each softmax group contains d nodes corresponding to the d words in the lexicon. Therefore, the RBM for each document is different, because the number of units depends on the length of the document. However, all the softmax groups within a document and across multiple documents share weights of their connections to the hidden units. The ith position in the document corresponds to the ith group of visible softmax units. The ith group of visible units is denoted by $v_i^{(1)} \ldots v_i^{(d)}$. The bias associated with $v_i^{(k)}$ is $b^{(k)}$. Note that the bias of the ith visible node depends only on k (word identity) and not on i (position of word in document). This is because the model uses a bag-of-words approach in which the positions of the words are irrelevant.

2. There are m hidden units denoted by $h_1 \ldots h_m$. The bias of the jth hidden unit is b_j.

3. Each hidden unit is connected to each of the $n_t \times d$ visible units. All softmax groups within a single RBM as well as across different RBMs (corresponding to different documents) share the same set of d weights. The kth hidden unit is connected to a group of d softmax units with a vector of d weights denoted by $\overline{W}^{(k)} = (w_1^{(k)} \ldots w_d^{(k)})$. In other words, the kth hidden unit *is connected to each of the n_t groups of d softmax units with the same set of weights $\overline{W}^{(k)}$.*

The architecture of the RBM is illustrated in Figure 6.8. Based on the architecture of the RBM, one can express the probabilities associated with the states of the hidden units with the use of the sigmoid function:

$$P(h_j = 1|\overline{v}^{(1)}, \ldots \overline{v}^{(d)}) = \frac{1}{1 + \exp(-b_j - \sum_{i=1}^{n_t} \sum_{k=1}^{d} v_i^{(k)} w_j^{(k)})} \tag{6.30}$$

One can also express the visible states with the use of the multinomial model:

$$P(v_i^{(k)} = 1|\overline{h}) = \frac{\exp(b^{(k)} + \sum_{j=1}^{m} w_j^{(k)} h_j)}{\sum_{l=1}^{d} \exp(b^{(l)} + \sum_{j=1}^{m} w_j^{(l)} h_j)} \tag{6.31}$$

The normalization factor in the denominator ensures that the sum of the probabilities of visible units over all the words always sums to 1. Furthermore, the right-hand side of the above equation is independent of the index i of the visible unit. This is because this model does not depend on the position of words in the document, and the modeling treats a document as a bag of words.

With these relationships, one can apply MCMC sampling to generate samples of the hidden and visible states for the contrastive divergence algorithm. Note that the RBMs are different for different documents, although these RBMs share weights. As in the case of the collaborative filtering application, each RBM is associated with only a single training example corresponding to the relevant document. The weight update used for gradient descent is the same as used for the traditional RBM. The only difference is that the weights across different visible units are shared. This approach is similar to what is performed in collaborative filtering. We leave the derivation of the weight updates as an exercise for the reader (see Exercise 5).

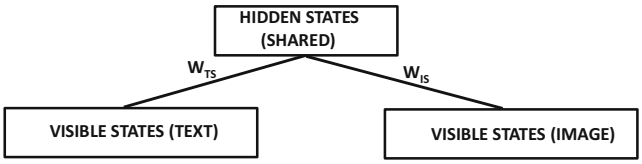
After the training has been performed, the reduced representation of each document is computed by applying Equation 6.30 to the words of a document. The real-valued value of the probabilities of the hidden units provides the m-dimensional reduced representation of the document. The approach described in this section is a simplification of a multilayer approach described in the original work [469].

6.5.5 RBMs for Machine Learning with Multimodal Data

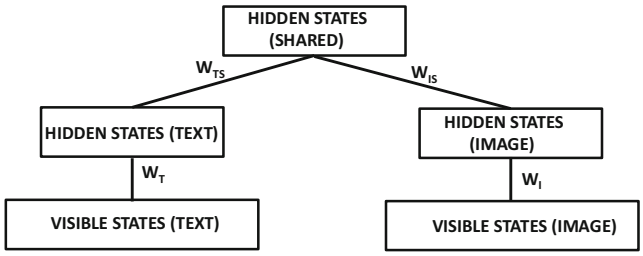
Boltzmann machines can also be used for machine learning with *multimodal* data. Multimodal data refers to a setting in which one is trying to extract information from data points with multiple modalities. For example, an image with a text description can be considered multimodal data. This is because this data object has both image and text modalities.

The main challenge in processing multimodal data is that it is often difficult to use machine learning algorithms on such heterogeneous features. Multimodal data is often processed by using a shared representation in which the two modes are mapped into a joint space. A common approach for this goal is *shared* matrix factorization. Numerous methods for using shared matrix factorization with text and image data are discussed in [6]. Since RBMs provide alternative representations to matrix factorization methods in many settings, it is natural to explore whether one can use this architecture to create a shared latent representation of the data.

An example [468] of an architecture for multimodal modeling is shown in Figure 6.9(a). In this example, it is assumed that the two modes correspond to text and image data. The image and the text data are used to create hidden states that are specific to images and



(a) A simple RBM for multimodal data



(b) A multimodal RBM with an added hidden layer

Figure 6.9: RBM architecture for multimodal data processing

text, respectively. These hidden states then feed into a single shared representation. The similarity of this architecture with the classification architecture of Figure 6.7 is striking. This is because both architectures try to map two types of features into a set of shared hidden states. These hidden states can then be used for different types of inference, such as using the shared representation for classification. As shown in Section 6.7, one can even enhance such unsupervised representations with backpropagation to fine-tune the approach. Missing data modalities can also be generated using this model.

One can optionally improve the expressive power of this model by using depth. An additional hidden layer has been added between the visible states and the shared representation in Figure 6.9(b). Note that one can add multiple hidden layers in order to create a deep network. However, we have not yet described how one can actually train a multilayer RBM. This issue is discussed in Section 6.7.

An additional challenge with the use of multimodal data is that the features are often not binary. There are several solutions to this issue. In the case of text (or data modalities with small cardinalities of discrete attributes), one can use a similar approach as used in the RBM for topic modeling where the count c of a discrete attribute is used to create c instances of the one-hot encoded attribute. The issue becomes more challenging when the data contains arbitrary real values. One solution is to discretize the data, although such an approach can lose useful information about the data. Another solution is to make changes to the energy function of the Boltzmann machine. A discussion of some of these issues is provided in the next section.

6.6 Using RBMs Beyond Binary Data Types

The entire discussion so far in this chapter has focussed on the use of RBMs for binary data types. Indeed the vast majority of RBMs are designed for binary data types. For some types of data, such as categorical data or ordinal data (e.g., ratings), one can use the softmax approach described in Section 6.5.2. For example, the use of softmax units for word-count data is discussed in Section 6.5.4. One can make the softmax approach work

with an ordered attribute, when the number of discrete values of that attribute is small. However, these methods are not quite as effective for real-valued data. One possibility is to use discretization in order to convert real-valued data into discrete data, which can be handled with softmax units. Using such an approach does have the disadvantage of losing a certain amount of representational accuracy.

The approach described in Section 6.5.2 does provide some hints about how different data types can be addressed. For example, categorical or ordinal data is handled *by changing the probability distribution* of visible units to be more appropriate to the problem at hand. In general, one might need to change the distribution of not only the visible units, but also the hidden units. This is because the nature of the hidden units is dependent on the visible units.

For real-valued data, a natural solution is to use Gaussian visible units. Furthermore, the hidden units are real-valued as well, and are assumed to contain a ReLU activation function. The energy for a particular combination $(\overline{v}, \overline{h})$ of visible and hidden units is given by the following:

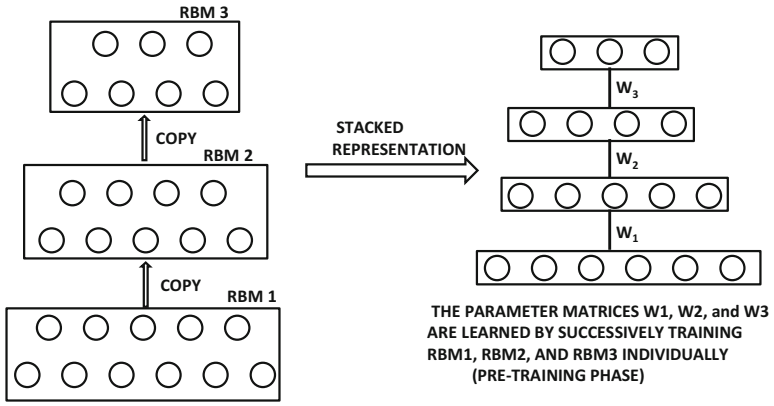
$$E(\overline{v}, \overline{h}) = \underbrace{\sum_i \frac{(v_i - b_i)^2}{2\sigma_i^2}}_{\text{Containment function}} - \sum_j b_j h_j - \sum_{i,j} \frac{v_i}{\sigma_i} h_j w_{ij} \tag{6.32}$$

Note that the energy contribution of the bias of visible units is given by a *parabolic containment function*. The effect of using this containment function is to keep the value of the ith visible unit close to b_i. As is the case for other types of Boltzmann machines, the derivatives of the energy function with respect to the different variables also provide the derivatives of the log-likelihoods. This is because the probabilities are always defined by exponentiating the energy function.

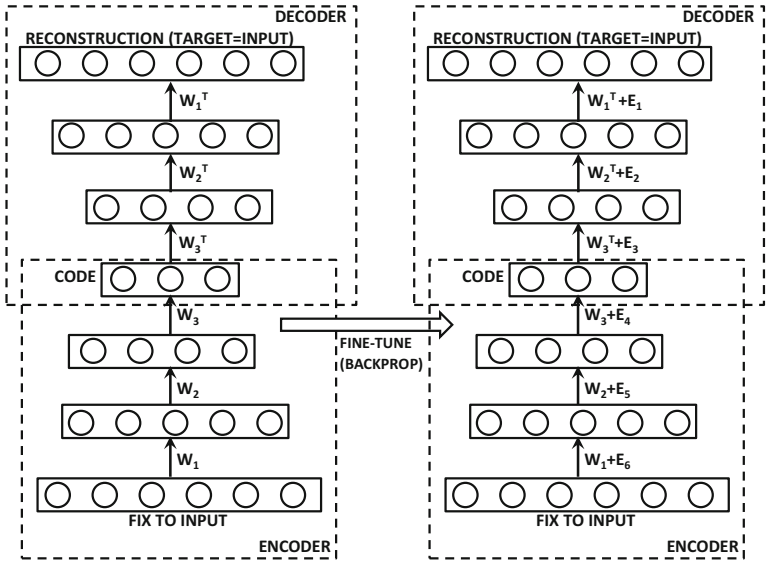
There are several challenges associated with the use of this approach. An important issue is that the approach is rather unstable with respect to the choice of the variance parameter σ. In particular, updates to the visible layer tend to be too small, whereas updates to the hidden layer tend to be too large. One natural solution to this dilemma is to use more hidden units than visible units. It is also common to normalize the input data to unit variance so that the standard deviation σ of the visible units can be set to 1. The ReLU units are modified to create a noisy version. Specifically, Gaussian noise with zero mean and variance $\log(1+\exp(v))$ is added to the value of the unit before thresholding it to nonnegative values. The motivation behind using such an unusual activation function is that it can be shown to be equivalent to a *binomial unit* [348, 495], which encodes more information than the binary unit that is normally used. It is important to enable this ability when working with real-valued data. The Gibbs sampling of the real-valued RBM is similar to a binary RBM, as are the updates to the weights once the MCMC samples are generated. It is important to keep the learning rates low to prevent instability.

6.7 Stacking Restricted Boltzmann Machines

Most of the power of conventional neural architectures arises from having multiple layers of units. Deeper networks are known to be more powerful, and can model more complex functions at the expense of fewer parameters. A natural question arises concerning whether similar goals can be achieved by putting together multiple RBMs. It turns out that the RBM is well suited to creating deep networks, and was used *earlier* than conventional neural networks for creating deep models with pretraining. In other words, the RBM is trained



(a) The stacked RBMs are trained sequentially in pretraining



(b) Pretraining is followed by fine-tuning with backpropagation

Figure 6.10: Training a multi-layer RBM

with Gibbs sampling, and the resulting weights are grandfathered into a conventional neural network with continuous sigmoid activations (instead of sigmoid-based discrete sampling). Why should one go through the trouble to train an RBM in order to train a conventional network at all? This is because of the fact that Boltzmann machines are trained in a fundamentally different way from the backpropagation approach in conventional neural networks. The contrastive divergence approach tends to train all layers jointly, which does not cause the same problems with the vanishing and exploding gradient problems, as is the case in conventional neural networks.

At first glance, the goal of creating deep networks from RBMs seems rather difficult. First, RBMs are not quite like feed-forward units that perform the computation in a particular direction. RBMs are symmetric models in which the visible units and hidden units are connected in the form of an undirected graphical model. Therefore, one needs to define a concrete way in which multiple RBMs interact with one another. In this context, a useful observation is that even though RBMs are symmetric and discrete models, the learned

weights can be used to define a related neural network that performs directed computation in the continuous space of activations. These weights are already quite close to the final solution because of how they have been learned with discrete sampling. Therefore, these weights can be fine-tuned with a relatively modest effort of traditional backpropagation. In order to understand this point, consider the single-layer RBM illustrated in Figure 6.4, which shows that even the single-layer RBM is equivalent to a directed graphical model of infinite length. However, once the visible states have been fixed, it suffices to keep only three layers of this computational graph, and perform the computations with the continuous values derived from sigmoid activations. This approach already provides a good approximate solution. The resulting network is a traditional autoencoder, although its weights have been (approximately) learned in a rather unconventional way. This section will show how this type of approach can also be applied to stacked RBMs.

What is a stacked set of RBMs? Consider a data set with d dimensions, for which the goal is to create a reduced representation with m_1 dimensions. One can achieve this goal with an RBM containing d visible units and m_1 hidden units. By training this RBM, one will obtain an m_1-dimensional representation of the data set. Now consider a second RBM that has m_1 visible units and m_2 hidden units. We can simply *copy* the m_1 outputs of the first RBM as the inputs to the second RBM, which has $m_1 \times m_2$ weights. As a result, one can train this new RBM to create an m_2-dimensional representation by using the outputs from the first RBM as its inputs. Note that we can repeat this process for k times, so that the last RBM is of size $m_{k-1} \times m_k$. Therefore, we *sequentially* train each of these RBMs by copying the output of one RBM into the input of another.

An example of a stacked RBM is shown on the left-hand side of Figure 6.10(a). This type of RBM is often shown with the concise diagram on the right-hand side of Figure 6.10(a). Note that the copying between two RBMs is a simple one-to-one copying between corresponding nodes, because the output layer of the rth RBM has exactly the same number of nodes as the input layer of the $(r+1)$th RBM. The resulting representations are *unsupervised* because they do not depend on a specific target. Another point is that the Boltzmann machine is an undirected model. However, by stacking the Boltzmann machine, we no longer have an undirected model because the upper layers receive feedback from the lower layers, but not vice versa. In fact, one can treat each Boltzmann machine as a single computational unit with many inputs and outputs, and the copying from one machine to another as the data transmission between two computational units. From this particular view of the stack of Boltzmann machines as a computational graph, it is even possible to perform backpropagation if one reverts to using the sigmoid units to create real-valued activations rather than to create the parameters needed for drawing binary samples. Although the use of real-valued activations is only an approximation, it already provides an excellent approximation because of the way in which the Boltzmann machine has been trained. This initial set of weights can be fine-tuned with backpropagation. After all, backpropagation can be performed on any computational graph, irrespective of the nature of the function computed inside the graph as long as a continuous function is computed. The fine tuning of backpropagation approach is particularly essential in the case of supervised learning, because the weights learned from a Boltzmann machine are always unsupervised.

6.7.1 Unsupervised Learning

Even in the case of unsupervised learning, the stacked RBM will generally provide reductions of better quality than a single RBM. However, the training of this RBM has to be performed carefully because results of high quality are not obtained by simply training all the layers

together. Better results are obtained by using a pretraining approach. Each of the three RBMs in Figure 6.10(a) are trained sequentially. First, RBM1 is trained using the provided training data as the values of the visible units. Then, the outputs of the first RBM are used to train RBM2. This approach is repeated to train RBM3. Note that one can greedily train as many layers as desired using this approach. Assume that the weight matrices for the three learned RBMs are W_1, W_2, and W_3, respectively. Once these weight matrices have been learned, one can put together an encoder-decoder pair with these three weight matrices as shown in Figure 6.10(b). The three decoders have weight matrices W_1^T, W_2^T, and W_3^T, because they perform the inverse operations of encoders. As a result, one now has a directed encoder-decoder network that can be trained with backpropagation like any conventional neural network. The states in this network are computed using directed probabilistic operations, rather than sampled with the use of Monte-Carlo methods. One can perform backpropagation through the layers in order to fine-tune the learning. Note that the weight matrices on the right-hand side of Figure 6.10(b) have been adjusted as a result of this fine tuning. Furthermore, the weight matrices of the encoder and the decoder are no longer related in a symmetric way as a result of the fine tuning. Such stacked RBMs provide reductions of higher quality compared to those with shallower RBMs [414], which is analogous to the behavior of conventional neural networks.

6.7.2 Supervised Learning

How can one learn the weights in such a way that the Boltzmann machine is encouraged to produce a particular type of output such a class labels? Imagine that one wants to perform a k-way classification with a stack of RBMs. The use of a single-layer RBM for classification has already been discussed in Section 6.5.3, and the corresponding architecture is illustrated in Figure 6.7. This architecture can be modified by replacing the single hidden layer with a stack of hidden layers. The final layer of hidden features are then connected to the visible softmax layer that outputs the k probabilities corresponding to the different classes. As in the case of dimensionality reduction, pretraining is helpful Therefore, the first phase is completely unsupervised in which the class labels are not used. In other words, we train the weights of each hidden layer separately. This is achieved by training the weights of the lower layers first and then the higher layers, as in any stacked RBM. After the initial weights have been set in an unsupervised way, one can perform the initial training of weights between the final hidden layer and visible layer of softmax units. One can then create a directed computational graph with these initial weights, as in the case of the unsupervised setting. Backpropagation is performed on this computational graph in order to perform fine tuning of the learned weights.

6.7.3 Deep Boltzmann Machines and Deep Belief Networks

One can stack the different layers of the RBM in various ways to achieve different types of goals. In some forms of stacking, the interactions between different Boltzmann machines are bi-directional. This variation is referred to as a *deep Boltzmann machine*. In other forms of stacking, some of the layers are uni-directional, whereas others are bi-directional. An example is a *deep belief network* in which only the upper RBM is bi-directional, whereas the lower layers are uni-directional. Some of these methods can be shown to be equivalent to various types of probabilistic graphical models like *sigmoid belief nets* [350].

A deep Boltzmann machine is particularly noteworthy because of the bi-directional connections between each pair of units. The fact that the copying occurs both ways means

that we can merge the nodes in adjacent nodes of two RBMs into a single layer of nodes. Furthermore, observe that one could rearrange the RBM into a bipartite graph by putting all the odd layers in one set and the even layers in another set. In other words, the deep RBM is equivalent to a single RBM. The difference from a single RBM is that the visible units form only a small subset of the units in one layer, and all pairs of nodes are not connected. Because of the fact that all pairs of nodes are not connected, the nodes in the upper layers tend to receive smaller weights than the nodes in the lower layers. As a result, pretraining again becomes necessary in which the lower layers are trained first, and then followed up with the higher layers in a greedy way. Subsequently, all layers are trained together in order to fine-tune the method. Refer to the bibliographic notes for details of these advanced models.

6.8 Summary

The earliest variant of the Boltzmann machine was the Hopfield network. The Hopfield network is an energy-based model, which stores the training data instances in its local minima. The Hopfield network can be trained with the Hebbian learning rule. A stochastic variant of the Hopfield network is the Boltzmann machine, which uses a probabilistic model to achieve greater generalization. Furthermore, the hidden states of the Boltzmann machine hold a reduced representation of the data. The Boltzmann machine can be trained with a stochastic variant of the Hebbian learning rule. The main challenge in the case of the Boltzmann machine is that it requires Gibbs sampling, which can be slow in practice. The restricted Boltzmann machine allows connections only between hidden nodes and visible nodes, which eases the training process. More efficient training algorithms are available for the restricted Boltzmann machine. The restricted Boltzmann machine can be used as a dimensionality reduction method; it can also be used in recommender systems with incomplete data. The restricted Boltzmann machine has also been generalized to count data, ordinal data, and real-valued data. However, the vast majority of RBMs are still constructed under the assumption of binary units. In recent years, several deep variants of the restricted Boltzmann machine have been proposed, which can be used for conventional machine learning applications like classification.

6.9 Bibliographic Notes

The earliest variant of the Boltzmann family of models was the Hopfield network [207]. The Storkey learning rule is proposed in [471]. The earliest algorithms for learning Boltzmann machines with the use of Monte Carlo sampling were proposed in [1, 197]. Discussions of Markov Chain Monte Carlo methods are provided in [138, 351], and many of these methods are useful for Boltzmann machines as well. RBMs were originally invented by Smolensky, and referred to as the harmonium. A tutorial on energy-based models is provided in [280]. Boltzmann machines are hard to train because of the interdependent stochastic nature of the units. The intractability of the partition function also makes the learning of the Boltzmann machine hard. However, one can estimate the partition function with *annealed importance sampling* [352]. A variant of the Boltzmann machine is the *mean-field Boltzmann machine* [373], which uses deterministic real units rather than stochastic units. However, the approach is a heuristic and hard to justify. Nevertheless, the use of real-valued approximations is popular at inference time. In other words, a traditional neural network

with real-valued activations and derived weights from the trained Boltzmann machine is often used for prediction. Other variations of the RBM, such as the neural autoregressive distribution estimator [265], can be viewed as autoencoders.

The efficient mini-batch algorithm for Boltzmann machines is described in [491]. The contrastive divergence algorithm, which is useful for RBMs, is described in [61, 191]. A variation referred to as *persistent contrastive divergence* is proposed in [491]. The idea of gradually increasing the value of k in CD_k over the progress of training was proposed in [61]. The work in [61] showed that even a single iteration of the Gibbs sampling approach (which greatly reduces burn-in time) produces only a small bias in the final result, which can be reduced by gradually increasing the value of k in CD_k over the course of training. This insight was key to the efficient implementation of the RBM. An analysis of the bias in the contrastive divergence algorithm may be found in [29]. The work in [479] analyzes the convergence properties of the RBM. It is also shows that the contrastive divergence algorithm is a heuristic, which does not really optimize any objective function. A discussion and practical suggestions for training Boltzmann machines may be found in [119, 193]. The universal approximation property of RBMs is discussed in [341].

RBMs have been used for a variety of applications like dimensionality reduction, collaborative filtering, topic modeling and classification. The use of the RBM for collaborative filtering is discussed in [414]. This approach is instructive because it also shows how one can use an RBM for categorical data containing a small number of values. The application of discriminative restricted Boltzmann machines to classification is discussed in [263, 264]. The topic modeling of documents with Boltzmann machines with softmax units (as discussed in the chapter) is based on [469]. Advanced RBMs for topic modeling with a Poisson distribution are discussed in [134, 538]. The main problem with these methods is that they are unable to work well with documents of varying lengths. The use of replicated softmax is discussed in [199]. This approach is closely connected to ideas from *semantic hashing* [415].

Most of the RBMs are proposed for binary data. However, in recent years, RBMs have also been generalized to other data types. The modeling of count data with softmax units is discussed in the context of topic modeling in [469]. The challenges associated with this type of modeling are discussed in [86]. The use of the RBM for the exponential distribution family is discussed in [522], and discussion for real-valued data is provided in [348]. The introduction of binomial units to encode more information than binary units was proposed in [495]. This approach was shown to be a noisy version of the ReLU [348]. The replacement of binary units with linear units containing Gaussian noise was first proposed in [124]. The modeling of documents with deep Boltzmann machines is discussed in [469]. Boltzmann machines have also been used for multimodal learning with images and text [357, 468].

Training of deep variations of Boltzmann machines provided the first deep learning algorithms that worked well [196]. These algorithms were the first pretraining methods, which were later generalized to other types of neural networks. A detailed discussion of pretraining may be found in Section 4.7 of Chapter 4. Deep Boltzmann machines are discussed in [417], and efficient algorithms are discussed in [200, 418].

Several architectures that are related to the Boltzmann machine provide different types of modeling capabilities. The Helmholtz machine and a wake-sleep algorithm are proposed in [195]. RBMs and their multilayer variants can be shown to be equivalent to different types of probabilistic graphical models such as sigmoid belief nets [350]. A detailed discussion of probabilistic graphical models may be found in [251]. In higher-order Boltzmann machines, the energy function is defined by groups of k nodes for $k > 2$. For example, an order-3 Boltzmann machine will contain terms of the form $w_{ijk}s_i s_j s_k$. Such higher-order machines

are discussed in [437]. Although these methods are potentially more powerful that traditional Boltzmann machines, they have not found much popularity because of the large amount of data they require to train.

6.10 Exercises

1. This chapter discusses how Boltzmann machines can be used for collaborative filtering. Even though discrete sampling of the contrastive divergence algorithm is used for learning the model, the final phase of inference is done using real-valued sigmoid and softmax activations. Discuss how you can use this fact to your advantage in order to fine-tune the learned model with backpropagation.

2. Implement the contrastive divergence algorithm of a restricted Boltzmann machine. Also implement the inference algorithm for deriving the probability distribution of the hidden units for a given test example. Use Python or any other programming language of your choice.

3. Consider a Boltzmann machine without a bipartite restriction (of the RBM), but with the restriction that all units are visible. Discuss how this restriction simplifies the training process of the Boltzmann machine.

4. Propose an approach for using RBMs for outlier detection.

5. Derive the weight updates for the RBM-based topic modeling approach discussed in the chapter. Use the same notations.

6. Show how you can extend the RBM for collaborative filtering (discussed in Section 6.5.2 of the chapter) with additional layers to make it more powerful.

7. A discriminative Boltzmann machine is introduced for classification towards the end of Section 6.5.3. However, this approach is designed for binary classification. Show how you can extend the approach to multi-way classification.

8. Show how you can modify the topic modeling RBM discussed in the chapter in order to create a hidden representation of each node drawn from a large, sparse graph (like a social network).

9. Discuss how you can enhance the model of Exercise 8 to include data about an unordered list of keywords associated with each node. (For example, social network nodes are associated with wall-post and messaging content.)

10. Discuss how you can enhance the topic modeling RBM discussed in the chapter with multiple layers.