Chapter 13

Structured Probabilistic Models for Deep Learning

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

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

The graphical models research community is large and has developed many different models, training algorithms, and inference algorithms. In this chapter, we provide basic background on some of the most central ideas of graphical models, with an emphasis on the concepts that have proven most useful to the deep learning research community. If you already have a strong background in graphical models, you may wish to skip most of this chapter. However, even a graphical model expert may benefit from reading the final section of this chapter, section 13.6, in which we highlight some of the unique ways that graphical models are used for deep learning algorithms. Deep learning practitioners tend to use very different model structures, learning algorithms, and inference procedures than are commonly used by the rest of the graphical models research community. In this chapter, we identify these differences in preferences and explain the reasons for them.

In this chapter we first describe the challenges of building large-scale probabilistic models in section 13.1. Next, we describe how to use a graph to describe the structure of a probability distribution in section 13.2. We then revisit the challenges we described in section 13.1 and show how the structured approach to probabilistic modeling can overcome these challenges in section 13.3. One of the major difficulties in graphical modeling is understanding which variables need to be able to interact directly, i.e., which graph structures are most suitable for a given problem. We outline two approaches to resolving this difficulty by learning about the dependencies in section 13.4. Finally, we close with a discussion of the unique emphasis that deep learning practitioners place on specific approaches to graphical modeling in section 13.6.

13.1 The Challenge of Unstructured Modeling

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

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

It is possible to ask probabilistic models to do many other tasks. These tasks are often more expensive than classification. Some of them require producing multiple output values. Most require a complete understanding of the entire structure of the input, with no option to ignore sections of it. These tasks include

- Density estimation: given an input $\boldsymbol{x}$, the machine learning system returns an estimate of $p(\boldsymbol{x})$. This requires only a single output, but it does require

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

a complete understanding of the entire input. If even one element of the vector is unusual, the system must assign it a low probability.

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

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

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

For an example of the sampling tasks on small natural images, see Fig. 13.1.

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

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

This is not feasible for several reasons:

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

- **Statistical efficiency**: As the number of parameters in a model increases, so does the amount of training examples needed to choose the values of those parameters using a statistical estimator. Because the table-based model has

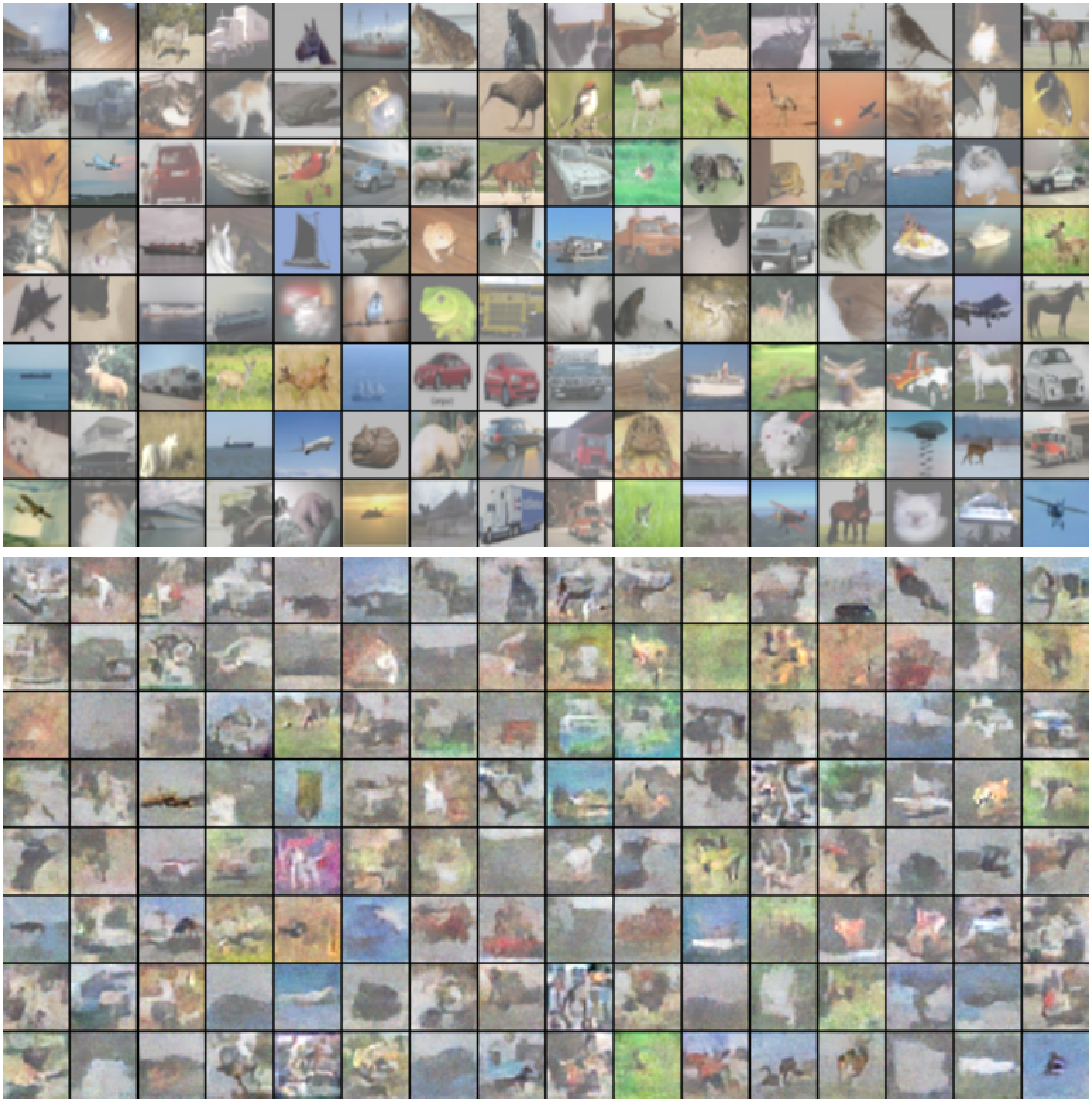

Figure 13.1: Probabilistic modeling of natural images. *Top:* Example 32 × 32 pixel color images from the CIFAR-10 dataset (Krizhevsky and Hinton, 2009). *Bottom:* Samples drawn from a structured probabilistic model trained on this dataset. Each sample appears at the same position in the grid as the training example that is closest to it in Euclidean space. This comparison allows us to see that the model is truly synthesizing new images, rather than memorizing the training data. Contrast of both sets of images has been adjusted for display. Figure reproduced with permission from (Courville *et al.*, 2011).

an astronomical number of parameters, it will require an astronomically large training set to fit accurately. Any such model will overfit the training set very badly.

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

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

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

For example, consider modeling the finishing times of a team in a relay race. Suppose the team consists of three runners, Alice, Bob, and Carol. At the start of the race, Alice carries a baton and begins running around a track. After completing her lap around the track, she hands the baton to Bob. Bob then runs his own lap and hands the baton to Carol, who runs the final lap. We can model each of their finishing times as a continuous random variable. Alice's finishing time does not depend on anyone else's, since she goes first. Bob's finishing time depends on Alice's, because Bob does not have the opportunity to start his lap until Alice has completed hers. If Alice finishes faster, Bob will finish faster, all else being equal. Finally, Carol's finishing time depends on both her teammates. If Alice is slow, Bob will probably finish late too, and Carol will have quite a late starting time and thus is likely to have a late finishing time as well. However, Carol's finishing time depends only *indirectly* on Alice's finishing time via Bob's. If we already know Bob's finishing time, we won't be able to estimate Carol's finishing time better by finding out what Alice's finishing time was. This means we can model the relay race using only two interactions: Alice's effect on Bob, and Bob's effect on Carol. We can omit the third, indirect interaction between Alice and Carol from our model.

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

13.2 Using Graphs to Describe Model Structure

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

There is more than one way to describe the interactions in a probability distribution using a graph. In the following sections we describe some of the most popular and useful approaches.

13.2.1 Directed Models

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

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

Let's continue with the relay race example from Section 13.1. Suppose we name Alice's finishing time t_0, Bob's finishing time t_1, and Carol's finishing time t_2. As we saw earlier, our estimate of t_1 depends on t_0. Our estimate of t_2 depends directly on t_1 but only indirectly on t_0. We can draw this relationship in a directed graphical model, illustrated in Fig. 13.2.

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

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

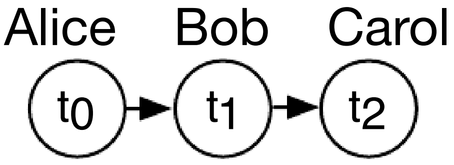

Figure 13.2: A directed graphical model depicting the relay race example. Alice's finishing time t_0 influences Bob's finishing time t_1, because Bob does not get to start running until Alice finishes. Likewise, Carol only gets to start running after Bob finishes, so Bob's finishing time t_1 influences Carol's finishing time t_2.

gives the parents of x_i in $\mathcal{G}$. The probability distribution over $\mathbf{x}$ is given by

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

In our relay race example, this means that, using the graph drawn in Fig. 13.2,

$$p(t_0, t_1, t_2) = p(t_0)p(t_1 \mid t_0)p(t_2 \mid t_1).$$

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

Suppose we represented time by discretizing time ranging from minute 0 to minute 10 into 6 second chunks. This would make t_0, t_1, and t_2 each be discrete variables with 100 possible values. If we attempted to represent $p(t_0, t_1, t_2)$ with a table, it would need to store 999,999 values (100 values of t_0 × 100 values of t_1 × 100 values of t_2, minus 1, since the probability of one of the configurations is made redundant by the constraint that the sum of the probabilities be 1). If instead, we only make a table for each of the conditional probability distributions, then the distribution over t_0 requires 99 values, the table defining t_1 given t_0 requires 9900 values, and so does the table defining t_2 and t_1. This comes to a total of 19,899 values. This means that using the directed graphical model reduced our number of parameters by a factor of more than 50!

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

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

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

13.2.2 Undirected Models

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

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

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

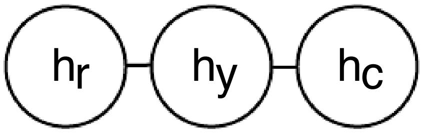

Figure 13.3: An undirected graph representing how your roommate's health h_r, your health h_y, and your work colleague's health h_c affect each other. You and your roommate might infect each other with a cold, and you and your work colleague might do the same, but assuming that your roommate and your colleague don't know each other, they can only infect each other indirectly via you.

model.

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

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

Let's call the random variable representing your health h_y, the random variable representing your roommate's health h_r, and the random variable representing your colleague's health h_c. See Fig. 13.3 for a drawing of the graph representing this scenario.

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

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

non-negative. Together they define an *unnormalized probability distribution*

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

The unnormalized probability distribution is efficient to work with so long as all the cliques are small. It encodes the idea that states with higher affinity are more likely. However, unlike in a Bayesian network, there is little structure to the definition of the cliques, so there is nothing to guarantee that multiplying them together will yield a valid probability distribution. See Fig. 13.4 for an example of reading factorization information from an undirected graph.

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

	$\mathrm{h}_y = 0$	$\mathrm{h}_y = 1$
$\mathrm{h}_c = 0$	2	1
$\mathrm{h}_c = 1$	1	10

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

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

13.2.3 The Partition Function

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

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

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

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

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

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

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

One important consideration to keep in mind when designing undirected models is that it is possible for Z not to exist. This happens if some of the variables in the model are continuous and the integral of $\tilde{p}$ over their domain diverges. For example, suppose we want to model a single scalar variable $\mathrm{x} \in \mathbb{R}$ with a single clique potential $\phi(x) = x^2$. In this case,

$$Z = \int x^2 dx.$$

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

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

$= 1)$ for $j \neq i$. Often, it is possible to

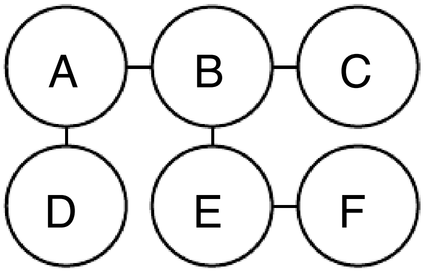

Figure 13.4: This graph implies that $p(A,B,C,D,E,F)$ can be written as $\frac{1}{Z}\phi_{A,B}(A,B)\phi_{B,C}(B,C)\phi_{A,D}(A,D)\phi_{B,E}(B,E)\phi_{E,F}(E,F)$ for an appropriate choice of the ϕ functions.

leverage the effect of a carefully chosen domain of a variable in order to obtain complicated behavior from a relatively simple set of ϕ functions. We'll explore a practical application of this idea later, in Chapter 20.7.

13.2.4 Energy-Based Models

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

$$\tilde{p}(\mathbf{x}) = \exp(-E(\mathbf{x})) \tag{13.1}$$

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

Any distribution of the form given by equation 13.1 is an example of a *Boltzmann distribution*. For this reason, many energy-based models are called *Boltzmann machines*. There is no accepted guideline for when to call a model an energy-based model and when to call it a Boltzmann machines. The term Boltzmann machine was first introduced to describe a model with exclusively binary variables, but today many models such as the mean-covariance restricted Boltzmann machine incorporate real-valued variables as well.

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

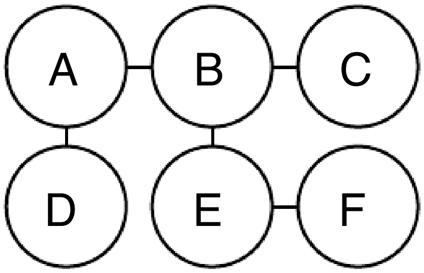

Figure 13.5: This graph implies that $E(\mathrm{a},\mathrm{b},\mathrm{c},\mathrm{d},\mathrm{e},\mathrm{f})$ can be written as $E_{\mathrm{a},\mathrm{b}}(\mathrm{a},\mathrm{b}) + E_{\mathrm{b},\mathrm{c}}(\mathrm{b},\mathrm{c}) + E_{\mathrm{a},\mathrm{d}}(\mathrm{a},\mathrm{d}) + E_{\mathrm{b},\mathrm{e}}(\mathrm{b},\mathrm{e}) + E_{\mathrm{e},\mathrm{f}}(\mathrm{e},\mathrm{f})$ for an appropriate choice of the per-clique energy functions. Note that we can obtain the ϕ functions in Fig. 13.4 by setting each ϕ to the exp of the corresponding negative energy, e.g., $\phi_{\mathrm{a},\mathrm{b}}(\mathrm{a},\mathrm{b}) = \exp(-E(\mathrm{a},\mathrm{b}))$.

Cliques in an undirected graph correspond to factors of the unnormalized probability function. Because $\exp(a)\exp(b) = \exp(a+b)$, this means that different cliques in the undirected graph correspond to the different terms of the energy function. In other words, an energy-based model is just a special kind of Markov network: the exponentiation makes each term in the energy function correspond to a factor for a different clique. See Fig. 13.5 for an example of how to read the form of the energy function from an undirected graph structure.

One part of the definition of an energy-based model serves no functional purpose from a machine learning point of view: the $-$ sign in Eq. 13.1. This $-$ sign could be incorporated into the definition of E, or for many functions E the learning algorithm could simply learn parameters with opposite sign. The $-$ sign is present primarily to preserve compatibility between the machine learning literature and the physics literature. Many advances in probabilistic modeling were originally developed by statistical physicists, for whom E refers to actual, physical energy and does not have arbitrary sign. Terminology such as “energy” and “partition function” remains associated with these techniques, even though their mathematical applicability is broader than the physics context in which they were developed. Some machine learning researchers (e.g., Smolensky (1986), who referred to negative energy as *harmony*) have chosen to emit the negation, but this is not the standard convention.

13.2.5 Separation and D-Separation

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

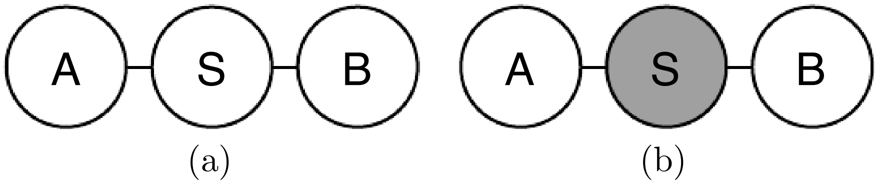

Figure 13.6: a) The path between random variable a and random variable b through s is active, because s is not observed. This means that a and b are not separated. b) Here s is shaded in, to indicate that it is observed. Because the only path between *ra* and b is through s, and that path is inactive, we can conclude that a and b are separated given s.

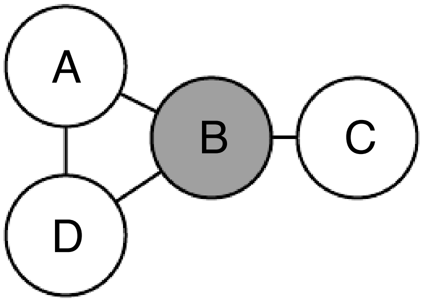

Figure 13.7: An example of reading separation properties from an undirected graph. Here b is shaded to indicate that it is observed. Because observing b blocks the only path from a to c, we say that a and c are separated from each other given b. The observation of b also blocks one path between a and d, but there is a second, active path between them. Therefore, a and d are not separated given b.

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

When we draw a graph, we can indicate observed variables by shading them in. See Fig. 13.6 for a depiction of how active and inactive paths in an undirected look when drawn in this way. See Fig. 13.7 for an example of reading separation from an undirected graph.

Similar concepts apply to directed models, except that in the context of directed models, these concepts are referred to as *d-separation.* The "d" stands for "dependence." D-separation for directed graphs is defined the same as separation for undirected graphs: We say that a set of variables $\mathbb{A}$ is d-separated from another set of variables $\mathbb{B}$ given a third set of variables $\mathbb{S}$ if the graph structure implies that $\mathbb{A}$ is independent from $\mathbb{B}$ given $\mathbb{S}$.

As with undirected models, we can examine the independences implied by the graph by looking at what active paths exist in the graph. As before, two variables are dependent if there is an active path between them, and d-separated if no such path exists. In directed nets, determining whether a path is active is somewhat more complicated. See Fig. 13.8 for a guide to identifying active paths in a directed model. See Fig. 13.9 for an example of reading some properties from a graph.

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

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

13.2.6 Converting between Undirected and Directed Graphs

In common parlance, we often refer to certain model classes as being undirected or directed. For example, we typically refer to RBMs as undirected and sparse coding as directed. This way of speaking can be somewhat leading, because no probabilistic model is inherently directed or undirected. Instead, some models are most easily *described* using a directed graph, or most easily described using an undirected graph.

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

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

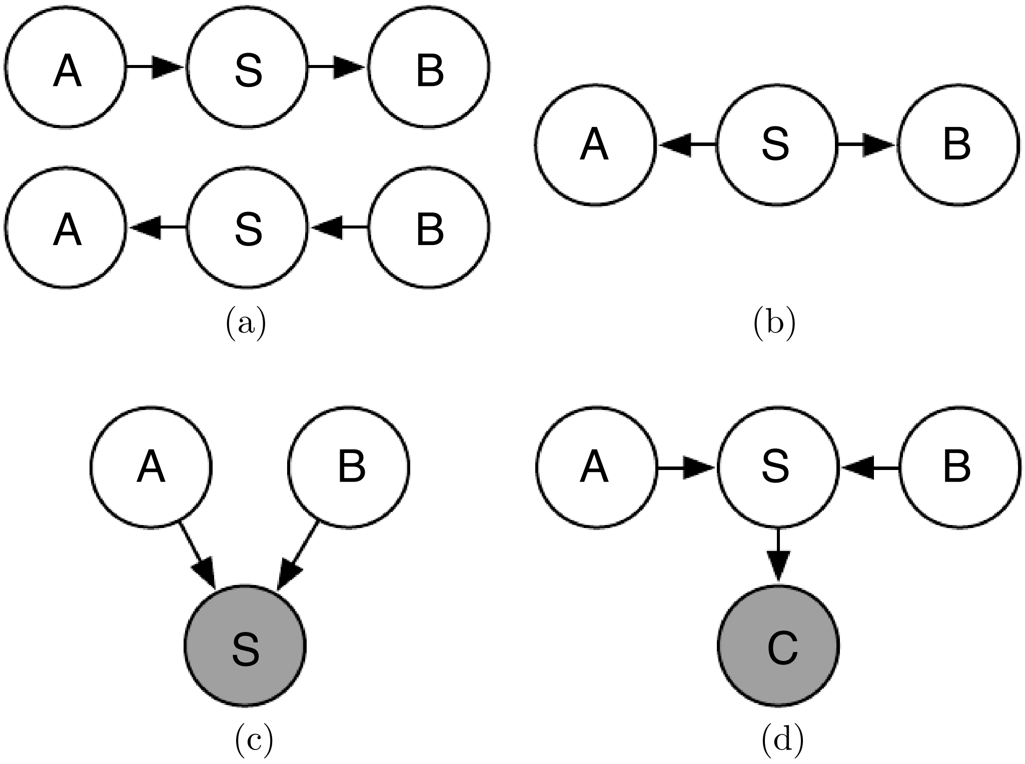

Figure 13.8: All of the kinds of active paths of length two that can exist between random variables a and *rb*. a) Any path with arrows proceeding directly from a to b or vice versa. This kind of path becomes blocked if s is observed. We have already seen this kind of path in the relay race example. b) a and b are connected by a *common cause* s. For example, suppose s is a variable indicating whether or not there is a hurricane and a and b measure the wind speed at two different nearby weather monitoring outposts. If we observe very high winds at station a, we might expect to also see high winds at b. This kind of path can be blocked by observing s. If we already know there is a hurricane, we expect to see high winds at b, regardless of what is observed at a. A lower than expected wind at a (for a hurricane) would not change our expectation of winds at b (knowing there is a hurricane). However, if s is not observed, then a and b are dependent, i.e., the path is inactive. c) a and b are both parents of s. This is called a *V-structure* or *the collider case*, and it causes a and a to be related by the *explaining away effect*. In this case, the path is actually active when s is observed. For example, suppose s is a variable indicating that your colleague is not at work. The variable a represents her being sick, while b represents her being on vacation. If you observe that she is not at work, you can presume she is probably sick or on vacation, but it's not especially likely that both have happened at the same time. If you find out that she is on vacation, this fact is sufficient to *explain* her absence, and you can infer that she is probably not also sick. d) The explaining away effect happens even if any descendant of s is observed! For example, suppose that c is a variable representing whether you have received a report from your colleague. If you notice that you have not received the report, this increases your estimate of the probability that she is not at work today, which in turn makes it more likely that she is either sick or on vacation. The only way to block a path through a V-structure is to observe none of the descendants of the shared child.

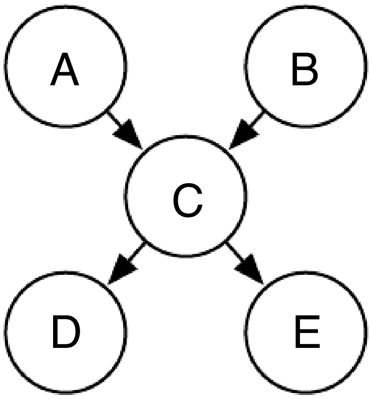

Figure 13.9

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

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

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

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

it does not imply any independences. TODO figure complete graph

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

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

Directed models are able to use one specific kind of substructure that undirected models cannot represent perfectly. This substructure is called an *immorality*. The structure occurs when two random variables a and b are both parents of a third random variable c, and there is no edge directly connecting a and b in either direction. (The name "immorality" may seem strange; it was coined in the graphical models literature as a joke about unmarried parents) To convert a directed model with graph $\mathcal{D}$ into an undirected model, we need to create a new graph $\mathcal{U}$. For every pair of variables x and y, we add an undirected edge connecting x and y to $\mathcal{U}$ if there is a directed edge (in either direction) connecting x and y in $\mathcal{D}$ or if x and y are both parents in $\mathcal{D}$ of a third variable z. The resulting $\mathcal{U}$ is known as a *moralized graph*. See Fig. 13.10 for examples of converting directed models to undirected models via moralization.

Likewise, undirected models can include substructures that no directed model can represent perfectly. Specifically, a directed graph $\mathcal{D}$ cannot capture all of the conditional independences implied by an undirected graph $\mathcal{U}$ if $\mathcal{U}$ contains a *loop* of length greater than three, unless that loop also contains a *chord*. A loop is a sequence of variables connected by undirected edges, with the last variable in the sequence connected back to the first variable in the sequence. A chord is a connection between any two non-consecutive variables in this sequence. If $\mathcal{U}$ has loops of length four or greater and does not have chords for these loops, we must add the chords before we can convert it to a directed model. Adding these chords discards some of the independence information that was encoded in $\mathcal{U}$. The graph formed by adding chords to $\mathcal{U}$ is known as a *chordal* or *triangulated* graph, because all the loops can now be described in terms of smaller, triangular loops. To build a directed graph $\mathcal{D}$ from the chordal graph, we need to also assign directions to the edges. When doing so, we must not create a directed cycle in $\mathcal{D}$, or the result does not define a valid directed probabilistic model. One way to assign directions to the edges in $\mathcal{D}$ is to impose an ordering on the random variables, then point each edge from the node that comes earlier in the ordering to the node that comes later in the ordering. TODO point to fig

IG HERE

TODO: started this above, need to scrap some some BNs encode indepen-

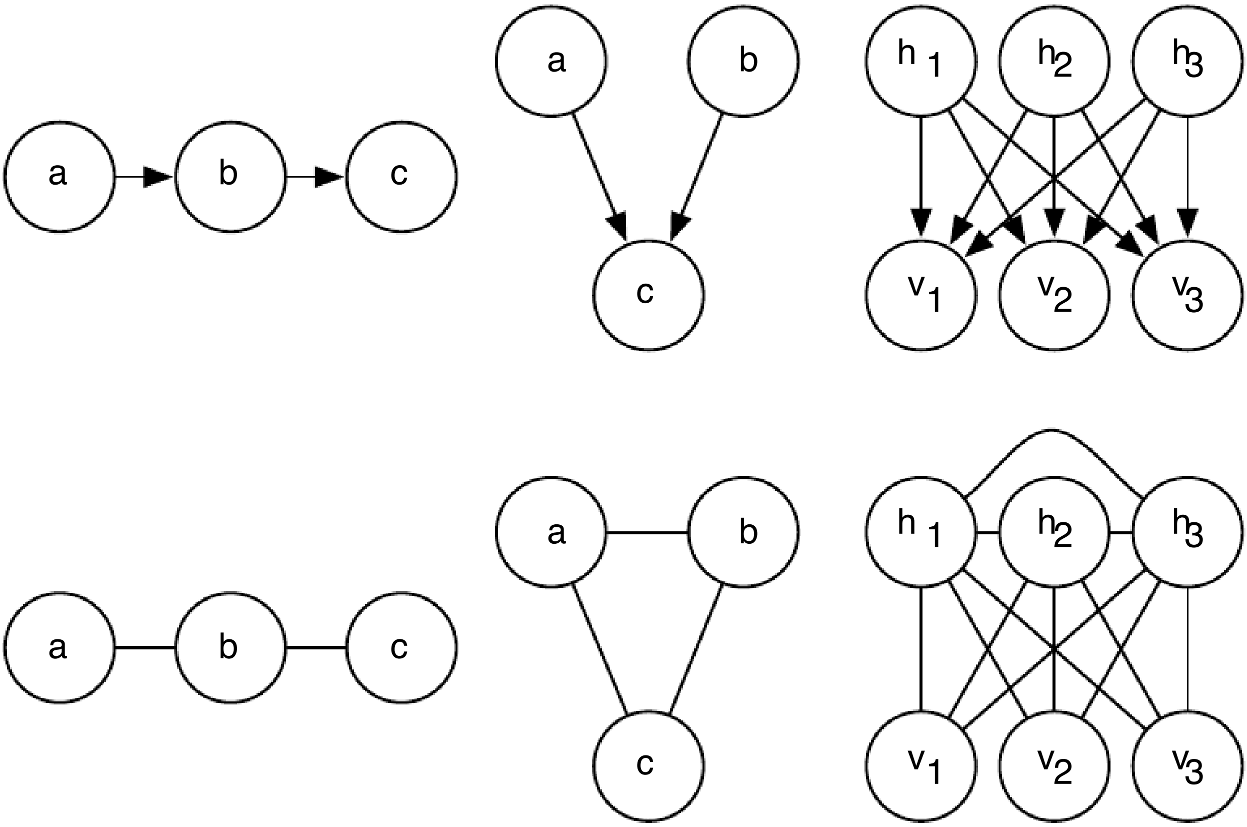

Figure 13.10: Examples of converting directed models to undirected models by constructing moralized graphs. *Left)* This simple chain can be converted to a moralized graph merely by replacing its directed edges with undirected edges. The resulting undirected model implies exactly the same set of independences and conditional independences. *Center)* This graph is the simplest directed model that cannot be converted to an undirected model without losing some independences. This graph consists entirely of a single immorality. Because a and b are parents of c, they are connected by an active path when c is observed. To capture this dependence, the undirected model must include a clique encompassing all three variables. This clique fails to encode the fact that a⊥b. *Right)* In general, moralization may add many edges to the graph, thus losing many implied independences. For example, this sparse coding graph requires adding moralizing edges between every pair of hidden units, thus introducing a quadratic number of new direct dependences.

dences that MNs can't encode, and vice versa example of BN that an MN can't encode: A and B are parents of C A is d-separated from B given the empty set The Markov net requires a clique over A, B, and C in order to capture the active path from A to B when C is observed This clique means that the graph cannot imply A is separated from B given the empty set example of a MN that a BN can't encode: A, B, C, D connected in a loop BN cannot have both A d-sep D given B, C and B d-sep C given A, D

In many cases, we may want to convert an undirected model to a directed model, or vice versa. To do so, we choose the graph in the new format that implies as many independences as possible, while not implying any independences that were not implied by the original graph.

To convert a directed model $\mathcal{D}$ to an undirected model $\mathcal{U}$, we re

TODO: conversion between directed and undirected models

13.2.7 Marginalizing Variables out of a Graph

TODO: marginalizing variables out of a graph

13.2.8 Factor Graphs

Factor graphs are another way of drawing undirected models that resolve an ambiguity in the graphical representation of standard undirected model syntax. In an undirected model, the scope of every ϕ function must be a subset of some clique in the graph. However, it is not necessary that there exist any ϕ whose scope contains the entirety of every clique. Factor graphs explicitly represent the scope of each ϕ function. Specifically, a factor graph is a graphical representation of an undirected model that consists of a bipartite undirected graph. Some of the nodes are drawn as circles. These nodes correspond to random variables as in a standard undirected model. The rest of the nodes are drawn as squares. These nodes correspond to the factors ϕ of the unnormalized probability distribution. Variables and factors may be connected with undirected edges. A variable and a factor are connected in the graph if and only if the variable is one of the arguments to the factor in the unnormalized probability distribution. No factor may be connected to another factor in the graph, nor can a variable be connected to a variable. See Fig. 13.11 for an example of how factor graphs can resolve ambiguity in the interpretation of undirected networks.

13.3 Advantages of Structured Modeling

TODO– note that we have already shown that some things are cheaper in the sections where we introduce the modeling syntax

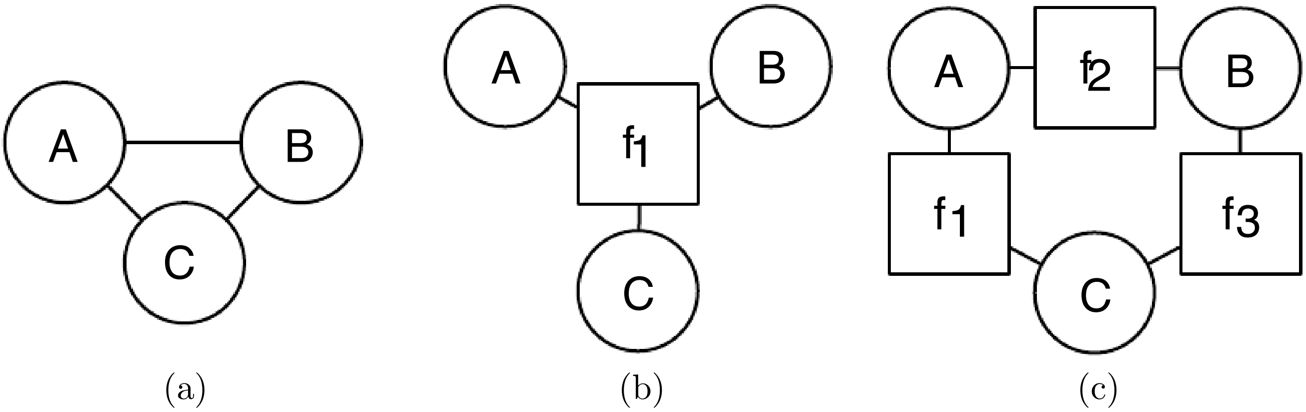

Figure 13.11: An example of how a factor graph can resolve ambiguity in the interpretation of undirected networks. a) An undirected network with a clique involving three variables a, b, and c. b) A factor graph corresponding to the same undirected model. This factor graph has one factor over all three variables. c) Another valid factor graph for the same undirected model. This factor graph has three factors, each over only two variables. Note that representation, inference, and learning are all asymptotically cheaper in (c) compared to (b), even though both require the same undirected graph to represent. TODO: make sure figure respects random variable notation

TODO: revisit each of the three challenges from sec:unstructured TODO: hammer point that graphical models convey information by leaving edges out TODO: need to show reduced cost of sampling, but first reader needs to know about ancestral and gibbs sampling.... TODO: benefit of separating representation from learning and inference

13.4 Learning about Dependencies

We describe here two types of random variables: observed or "visible" variables **v** and latent or "hidden" variables **h**. The observed variables **v** correspond to the variables actually provided in the data set during training. **h** consists of variables that are introduced to the model in order to help it explain the structure in **v**. Generally the exact semantics of **h** depend on the model parameters and are created by the learning algorithm. The motivation for this is twofold.

13.4.1 Latent Variables Versus Structure Learning

Often the different elements of **v** are highly dependent on each other. A good model of **v** which did not contain any latent variables would need to have very large numbers of parents per node in a Bayesian network or very large cliques in a Markov network. Just representing these higher order interactions is costly–both in a computational sense, because the number of parameters that must be stored in memory scales exponentially with the number of members in a clique, but also in a statistical sense, because this exponential number of parameters requires a wealth of data to estimate accurately.

There is also the problem of learning which variables need to be in such large cliques. An entire field of machine learning called *structure learning* is devoted to this problem . For a good reference on structure learning, see (Koller and Friedman, 2009). Most structure learning techniques are a form of greedy search. A structure is proposed, a model with that structure is trained, then given a score. The score rewards high training set accuracy and penalizes model complexity. Candidate structures with a small number of edges added or removed are then proposed as the next step of the search, and the search proceeds to a new structure that is expected to increase the score.

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

13.4.2 Latent Variables for Feature Learning

Another advantage of using latent variables is that they often develop useful semantics.

As discussed in section 3.10.6, the mixture of Gaussians model learns a latent variable that corresponds to which category of examples the input was drawn from. This means that the latent variable in a mixture of Gaussians model can be used to do classification.

In Chapter 15 we saw how simple probabilistic models like sparse coding learn latent variables that can be used as input features for a classifier, or as coordinates along a manifold. Other models can be used in this same way, but deeper models and models with different kids of interactions can create even richer descriptions of the input. Most of the approaches mentioned in sec. 13.4.2 accomplish feature learning by learning latent variables. Often, given some model of $\mathbf{v}$ and $\mathbf{h}$, it turns out that $\mathbb{E}[\mathbf{h} \mid \mathbf{v}]$ TODO: uh-oh, is there a collision between set notation and expectation notation? or $\text{argmax}_{\boldsymbol{h}} p(\boldsymbol{h}, \boldsymbol{v})$ is a good feature mapping for $\boldsymbol{v}$.

TODO: appropriate links to Monte Carlo methods chapter spun off from here

13.5 Inference and Approximate Inference over Latent Variables

As soon as we introduce latent variables in a graphical model, this raises the question: how to choose values of the latent variables $\boldsymbol{h}$ given values of the visible variables $\boldsymbol{x}$? This is what we call *inference*, in particular inference over the latent variables. The general question of inference is to guess some variables given others.

TODO: inference has definitely been introduced above... TODO: mention loopy BP, show how it is very expensive for DBMs

TODO: briefly explain what variational inference is and reference approximate inference chapter

13.5.1 Reparametrization Trick

Sometimes, in order to estimate the stochastic gradient of an expected loss over some random variable $\mathbf{h}$, with respect to parameters that influence $\mathbf{h}$, we would like to compute gradients through $\mathbf{h}$, i.e., on the parameters that influenced the probability distribution from which $\mathbf{h}$ was sampled. If $\mathbf{h}$ is continuous-valued, this is generally possible by using the *reparametrization trick*, i.e., rewriting

$$\mathbf{h} \sim p(\mathbf{h} \mid \theta) \tag{13.2}$$

as

$$\mathbf{h} = f(\theta, \eta) \tag{13.3}$$

where η is some independent noise source of the appropriate dimension with density $p(\eta)$, and f is a continuous (differentiable almost everywhere) function. The reparametrization trick is the idea that if the random variable to be integrated over is continuous, we can *back-propagate* through the process that gave rise to it in order to figure how to change that process.

For example, let us suppose we want to estimate the expected gradient

$$\frac{\partial}{\partial\theta} \int L(\mathbf{h}) p(\mathbf{h} \mid \theta) d\mathbf{h} \tag{13.4}$$

where the parameters θ influences the random variable $\mathbf{h}$ which in term influence our loss L. A very efficient (Kingma and Welling, 2014b; Rezende *et al.*, 2014) way to achieve[6] this is to perform the reparametrization in Eq. 13.3 and the corresponding change of variable in the integral of Eq. 13.4, integrating over η rather than $\mathbf{h}$:

$$\frac{\partial}{\partial\theta} \int L(f(\theta, \eta)) p(eta) d\eta. \tag{13.5}$$

[6] compared to approaches that do not back-propagate through the generation of $\mathbf{h}$

We can now more easily enter the derivative in the integral, getting

$$g = \int \frac{\partial L(f(\theta, \eta))}{\partial \theta} p(eta) d\eta.$$

Finally, we get a stochastic gradient estimator

$$\hat{g} = \frac{\partial L(f(\theta, \eta))}{\partial \theta}$$

where we sampled $\eta \sim p(\eta)$ and $E[\hat{g}] = g$.

This trick was first introduced to machine learning by Williams (1992) in the context of back-propagation. The idea of back-propagating through a sampling process was independently re-discovered and re-popularized in the context of deep generative models more recently by Bengio (2013b); Bengio *et al.* (2013a); Kingma (2013). The underlying principle was actually discovered much earlier in the information theory community (Price, 1958; Bonnet, 1964). The theorems from Price (1958); Bonnet (1964) were first used in a variational context by Opper and Archambeau (2009); Challis and Barber (2012) and for efficient Bayesian inference by Salimans and Knowles (2013). It was further developed and analyzed by Kingma and Welling (2014b) and was used to train generative stochastic networks (GSNs) (Bengio *et al.*, 2014a,b), described in Section 20.12, which can be viewed as recurrent networks with noise injected both in input and hidden units (with each time step corresponding to one step of a generative Markov chain). The reparametrization trick was also used to estimate the parameter gradient in variational auto-encoders (Kingma and Welling, 2014a; Rezende *et al.*, 2014; Kingma *et al.*, 2014), which are described in Section 20.9.3.

13.6 The Deep Learning Approach to Structured Probabilistic Models

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

The most striking difference between the deep learning style of graphical model design and the traditional style of graphical model design is that the deep learning style heavily emphasizes the use of latent variables. Deep learning models typically have more latent variables than observed variables. Moreover, the practitioner typically does not intend for the latent variables to take on any specific semantics ahead of time— the training algorithm is free to invent the concepts it needs to model a particular dataset. The latent variables are usually not very easy for a human to interpret after the fact, though visualization techniques may allow some rough characterization of what they represent. Complicated non-linear interactions between variables are accomplished via indirect connections that flow through multiple latent variables. By contrast, traditional graphical models usually contain variables that are at least occasionally observed, even if many of the variables are missing at random from some training examples. Complicated non-linear interactions between variables are modeled by using higher-order terms, with structure learning algorithms used to prune connections and control model capacity. When latent variables are used, they are often designed with some specific semantics in mind—the topic of a document, the intelligence of a student, the disease causing a patient's symptoms, etc. These models are often much more interpretable by human practitioners and often have more theoretical guarantees, yet are less able to scale to complex problems and are not reuseable in as many different contexts as deep models.

Another obvious difference is the kind of graph structure typically used in the deep learning approach. This is tightly linked with the choice of inference algorithm. Traditional approaches to graphical models typically aim to maintain the tractability of exact inference. When this constraint is too limiting, a popular exact inference algorithm is loopy belief propagation. Both of these approaches often work well with very sparsely connected graphs. By comparison, very few interesting deep models admit exact inference, and loopy belief propagation is almost never used for deep learning. Most deep models are designed to make Gibbs sampling or variational inference algorithms, rather than loopy belief propagation, efficient. Another consideration is that deep learning models contain a very large number of latent variables, making efficient numerical code essential. As a result of these design constraints, most deep learning models are organized into regular repeating patterns of units grouped into layers, but neighboring layers may be fully connected to each other. When sparse connections are used, they usually follow a regular pattern, such as the block connections used in convolutional models.

Finally, the deep learning approach to graphical modeling is characterized by a marked tolerance of the unknown. Rather than simplifying the model until all quantities we might want can be computed exactly, we increase the power of the model until it is just barely possible to train or use. We often use models whose marginal distributions cannot be computed, and are satisfied simply to draw approximate samples from these models. We often train models with an intractable objective function that we cannot even approximate in a reasonable amount of time, but we are still able to approximately train the model if we can efficiently obtain an estimate of the gradient of such a function. The deep learning

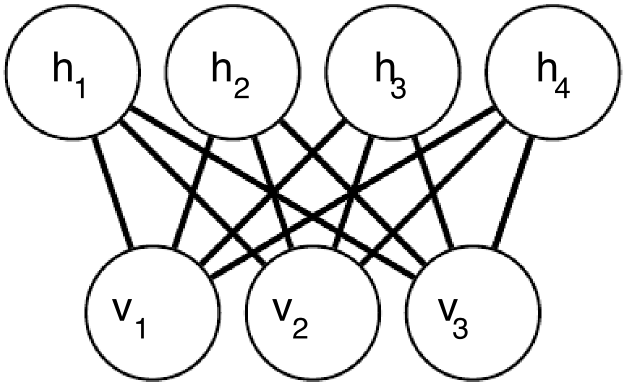

Figure 13.12: An example RBM drawn as a Markov network

approach is often to figure out what the minimum amount of information we absolutely need is, and then to figure out how to get a reasonable approximation of that information as quickly as possible.

13.6.1 Example: The Restricted Boltzmann Machine

TODO: rework this section. Add pointer to Chapter 20.2. TODO what do we want to exemplify here?

The *restricted Boltzmann machine* (RBM) (Smolensky, 1986) or *harmonium* is an example of a model that TODO what do we want to exemplify here?

It is an energy-based model with binary visible and hidden units. Its energy function is

$$E(v, h) = -b^\top v - c^\top h - v^\top W h$$

where $\boldsymbol{b}$, $\boldsymbol{c}$, and $\boldsymbol{W}$ are unconstrained, real-valued, learnable parameters. The model is depicted graphically in Fig. 13.12. As this figure makes clear, an important aspect of this model is that there are no direct interactions between any two visible units or between any two hidden units (hence the "restricted," a general Boltzmann machine may have arbitrary connections).

The restrictions on the RBM structure yield the nice properties

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

and

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

The individual conditionals are simple to compute as well, for example

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

Together these properties allow for efficient block Gibbs sampling, alternating between sampling all of $\mathbf{h}$ simultaneously and sampling all of $\mathbf{v}$ simultaneously.

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

$$\frac{\partial}{\partial \boldsymbol{W}_{i,j}} \mathbb{E}_{\mathbf{v},\mathbf{h}} E(\mathbf{v}, \mathbf{h}) = -\mathrm{v}_i \mathrm{h}_j .$$

These two properties–efficient Gibbs sampling and efficient derivatives– make it possible to train the RBM with stochastic approximations to $\nabla_\theta \log Z$.

13.6.2 The Computational Challenge with High-Dimensional Distributions

TODO: this whole section should probably just be cut, IG thinks YB has written the same thing in 2-3 other places (ml.tex for sure, and maybe also manifolds.tex and prob.tex, possibly others IG hasn't read yet) YB doesn't seem to have read the intro part of this chapter which discusses these things in more detail, double check to make sure there's not anything left out above If this section is kept, it needs cleanup, i.e. a instead A, etc. If this section is cut, need to search for refs to it and move them to one of the other versions of it

High-dimensional random variables actually bring two challenges: a statistical challenge and a computational challenge.

The *statistical challenge* was introduced in Section 5.12 and regards generalization: the number of configurations we may want to distinguish can grow exponentially with the number of dimensions of interest, and this quickly becomes much larger than the number of examples one can possibly have (or use with bounded computational resources).

The *computational challenge* associated with high-dimensional distributions arises because many algorithms for learning or using a trained model (especially those based on estimating an explicit probability function) involve intractable computations that grow exponentially with the number of dimensions.

With probabilistic models, this computational challenge arises because of intractable sums (summing over an exponential number of configurations) or intractable maximizations (finding the best out of an intractable number of configurations), discussed mostly in the third part of this book.

- **Intractable inference**: inference is discussed mostly in Chapter 19. It regards the question of guessing the probable values of some variables A, given other variables B, with respect to a model that captures the joint distribution between A, B and C. In order to even compute such conditional probabilities one needs to sum over the values of the variables C, as well as compute a normalization constant which sums over the values of A and C.

- **Intractable normalization constants (the partition function)**: the partition function is discussed mostly in Chapter 18. Normalizing constants of probability functions come up in inference (above) as well as in learning. Many probabilistic models involve such a constant. Unfortunately, the parameters (which we want to tune) influence that constant, and computing the gradient of the partition function with respect to the parameters is generally as intractable as computing the partition function itself. Monte-Carlo Markov chain (MCMC) methods (Chapter 14) are often used to deal with the partition function (computing it or its gradient) but they may also suffer from the curse of dimensionality, when the number of modes of the distribution of interest is very large, and these modes are well separated (Section 14.2).

One way to confront these intractable computations is to approximate them, and many approaches have been proposed, discussed in the chapters listed above. Another interesting way would be to avoid these intractable computations altogether by design, and methods that do not require such computations are thus very appealing. Several generative models based on auto-encoders have been proposed in recent years, with that motivation, and are discussed at the end of Chapter 20.