An overview of exact inference in graphical models

Tom Minka

February 10, 2005

1 Introduction

Suppose we have five variables (A, B, C, D, E) whose distribution is defined as the product of six pairwise potentials:

$$p(A, B, C, D, E) = f_{AB}(A, B) f_{BC}(B, C) f_{AC}(A, C) f_{AD}(A, D) f_{AE}(A, E) f_{DE}(D, E) \tag{1}$$

A typical inference task is to compute the normalizer of this distribution, i.e.

$$Z = \sum_{ABCDE} p(A, B, C, D, E) \tag{2}$$

Because of the structure of the distribution, we can rewrite this huge summation in terms of smaller sums, for example:

$$Z = \sum_A (\sum_B f_{AB} \sum_C f_{AC} f_{BC})(\sum_D f_{AD} \sum_E f_{AE} f_{DE}) \tag{3}$$

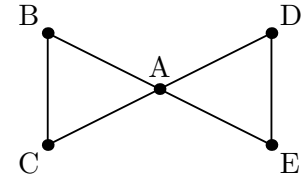
(For compactness we are writing $f_{AB}(A, B)$ as simply f_{AB}.) This is the basic idea behind every algorithm for exact inference in graphical models.

2 Markov networks and join trees

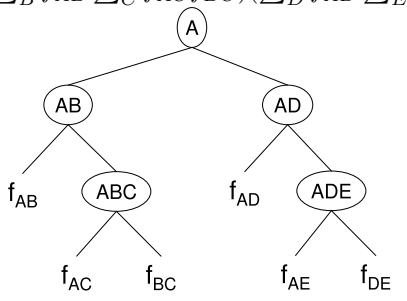
The transformation from (2) to (3) can be represented in terms of graphs. The distribution (1) can be represented by a factor graph. For our purposes, we can use an even simpler representation called a **Markov network**, which joins two variables with an edge if they appear together in any factor. Figure 1(c) gives the Markov network corresponding to (1).

The summation (3) can be represented by a **join tree**, shown in figure 1(d). The interior nodes of the tree are called **clusters**, and represent multiplications. The edges between clusters are called **separators**, and represent

(a) $Z = \sum_{ABCDE} f_{AB} f_{BC} f_{AC} f_{AD} f_{AE} f_{DE}$ (b) $Z = \sum_A (\sum_B f_{AB} \sum_C f_{AC} f_{BC})(\sum_D f_{AD} \sum_E f_{AE} f_{DE})$



(c)



(d)

Figure 1: (a) The normalizer of a factored distribution. (c) A Markov network describing the factors. (b) An efficient representation of the normalizer. (d) A join tree corresponding to (b).

summations. The clusters are labeled with the set of variables involved in the multiplication. This is only for convenience, since the labels are completely defined by the structure of the tree.

The question is how to produce an efficient join tree from a Markov network. There are a variety of ways to do this, the most common is the **variable elimination method**. In this method, you first pick the order in which variables are to be summed out. For example, in figure 1 the ordering would be (C, B, E, D, A) or (E, D, C, B, A). The join tree then comes from merging factors bottom-up:

1. Start with all factors in a pool.
2. Taking each variable in turn:
   - Find all of the factors in the pool that mention the variable and create a cluster in the join tree which parents them all.
   - Remove those factors from the pool, and insert the new cluster. (The factor pool can be implemented efficiently by sorting the factors into *buckets*.)

At the end of the algorithm, the pool will contain only one cluster, corresponding to the last eliminated variable.

The variable order determines the complexity of the join tree. Thus the problem becomes one of choosing a good variable order for a Markov network. One widely-used heuristic is to greedily pick the variable with fewest neighbors, removing it from the Markov net while linking all of its neighbors (to simulate clustering in the join tree).

There are also methods to construct a join tree from top-down, by repeatedly partitioning the Markov network. Some of these algorithms are guaranteed to produce trees close to the optimal tree. However, they are much more expensive than variable elimination.

A **junction tree** is a condensed version of a join tree, which hides clusters that are subsets of other clusters. Factors are always subsets of some cluster, so they are always hidden. The junction tree for figure 1 is simply the two clusters ABC and ADE, with an edge between them.

3 Inference in join trees

The join tree defines an efficient decomposition of the summation task—it does not specify how the computation should be carried out. To actually compute Z, there are two main approaches, corresponding to processing the tree bottom-up or top-down.

The **variable elimination method** processes the tree bottom-up. It works similarly to the method for producing the tree. We multiply a set of factors together, creating a new, bigger factor on the domain of the entire cluster, sum out one variable, and then insert the new factor into the pool. For the example in figure 1, the

order of operations would be:

$$t_{ABC}(A,B,C) = f_{AC}(A,C)f_{BC}(B,C) \tag{4}$$
$$t_{ABC\to AB}(A,B) = \sum_C t_{ABC}(A,B,C) \tag{5}$$
$$t_{AB}(A,B) = f_{AB}(A,B)t_{ABC\to AB}(A,B) \tag{6}$$
$$t_{AB\to A}(A) = \sum_B t_{AB}(A,B) \tag{7}$$
$$t_{ADE}(A,D,E) = f_{AE}(A,E)f_{DE}(D,E) \tag{8}$$
$$t_{ADE\to AD}(A,D) = \sum_E t_{ADE}(A,D,E) \tag{9}$$
$$t_{AD}(A,D) = f_{AD}(A,D)t_{ADE\to AD}(A,D) \tag{10}$$
$$t_{AD\to A}(A) = \sum_D t_{AD}(A,D) \tag{11}$$
$$t_A(A) = t_{AB\to A}(A)t_{AD\to A}(A) \tag{12}$$
$$Z = \sum_A t_A(A) \tag{13}$$

The temporaries t_{ABC} correspond to clusters, and $t_{ABC\to AB}$ correspond to separators. If all variables have K states, then the memory requirement is $K^3 + K$, because when we reach line (8) we have to simultaneously store $t_{AB\to A}(A)$ and $t_{ADE}(A,D,E)$.

To save memory, the tree can instead be processed top-down, using the method of **recursive conditioning.** This method is best understood from the summation representation (3). Instead of working from the inside-out, you work from the outside-in. You loop over the values of the outermost variable (A), recursively compute the value of the child expressions, multiply the results from each child, and sum. From the join tree perspective, it corresponds to a depth-first traversal, where at each cluster you loop over the "new" variables in the cluster, i.e. the variables in the cluster that do not appear in its parent.

The number of additions and multiplies is the same for both methods. But for the join tree in figure 1, the memory requirement of recursive conditioning is constant, independent of K. This is not true for all join trees, however. In figure 1, each cluster is a superset of its parent, so there is no redundancy in the recursive calls. In the general case, a cluster may not depend on some variables in its parent, which means some recursive calls are redundant, i.e. they differ in variables which the cluster does not depend on. To catch these redundancies, we need to memoize the results of each call, requiring additional storage equivalent to variable elimination. But this is only needed for a subset of the clusters, so the overall memory requirement is less than for variable elimination. Also, you can choose not to memoize all of the results, trading some extra time for less space.

Another advantage of recursive conditioning is that it can exploit context-sensitive independencies, i.e. structure not represented by the factor graph alone. However, by switching to a more detailed representation of join trees, called **case-factor diagrams**, we can exploit the additional structure while using either top-down or bottom-up inference.

4 Optimization

Now consider other inference tasks. One closely related task is to compute the highest probability of all states:

$$\max_{ABCDE} p(A, B, C, D, E) \tag{14}$$

This task can be decomposed in exactly the same way as (3), with "max" replacing "sum". We can find the maximum by either top-down or bottom-up processing on the join tree. Bottom-up corresponds to the Viterbi algorithm, while top-down corresponds to a depth-first search. Top-down again has an advantage in terms of memory, and can incorporate heuristics to speed up the search. For example, each subtree could compute a cheap upper bound to the value of the maximum it could achieve. Then you only need to recurse into subtrees whose upper bound is higher than the best value found so far.