26

Exact Marginalization in Graphs

We now take a more general view of the tasks of inference and marginalization. Before reading this chapter, you should read about message passing in Chapter 16.

▶ 26.1 The general problem

Assume that a function P^* of a set of N variables $\mathbf{x} \equiv \{x_n\}_{n=1}^N$ is defined as a product of M *factors* as follows:

$$P^*(\mathbf{x}) = \prod_{m=1}^{M} f_m(\mathbf{x}_m). \tag{26.1}$$

Each of the factors $f_m(\mathbf{x}_m)$ is a function of a subset $\mathbf{x}_m$ of the variables that make up $\mathbf{x}$. If P^* is a positive function then we may be interested in a second normalized function,

$$P(\mathbf{x}) \equiv \tfrac{1}{Z} P^*(\mathbf{x}) = \tfrac{1}{Z} \prod_{m=1}^{M} f_m(\mathbf{x}_m), \tag{26.2}$$

where the normalizing constant Z is defined by

$$Z = \sum_{\mathbf{x}} \prod_{m=1}^{M} f_m(\mathbf{x}_m). \tag{26.3}$$

As an example of the notation we've just introduced, here's a function of three binary variables x_1, x_2, x_3 defined by the five factors:

$$\begin{aligned}
f_1(x_1) &= \begin{cases} 0.1 & x_1 = 0 \\ 0.9 & x_1 = 1 \end{cases} \\
f_2(x_2) &= \begin{cases} 0.1 & x_2 = 0 \\ 0.9 & x_2 = 1 \end{cases} \\
f_3(x_3) &= \begin{cases} 0.9 & x_3 = 0 \\ 0.1 & x_3 = 1 \end{cases} \\
f_4(x_1, x_2) &= \begin{cases} 1 & (x_1, x_2) = (0,0) \text{ or } (1,1) \\ 0 & (x_1, x_2) = (1,0) \text{ or } (0,1) \end{cases} \\
f_5(x_2, x_3) &= \begin{cases} 1 & (x_2, x_3) = (0,0) \text{ or } (1,1) \\ 0 & (x_2, x_3) = (1,0) \text{ or } (0,1) \end{cases} \\
P^*(\mathbf{x}) &= f_1(x_1) f_2(x_2) f_3(x_3) f_4(x_1, x_2) f_5(x_2, x_3) \\
P(\mathbf{x}) &= \tfrac{1}{Z} f_1(x_1) f_2(x_2) f_3(x_3) f_4(x_1, x_2) f_5(x_2, x_3).
\end{aligned} \tag{26.4}$$

The five subsets of $\{x_1, x_2, x_3\}$ denoted by $\mathbf{x}_m$ in the general function (26.1) are here $\mathbf{x}_1 = \{x_1\}$, $\mathbf{x}_2 = \{x_2\}$, $\mathbf{x}_3 = \{x_3\}$, $\mathbf{x}_4 = \{x_1, x_2\}$, and $\mathbf{x}_5 = \{x_2, x_3\}$.

The function $P(\mathbf{x})$, by the way, may be recognized as the posterior probability distribution of the three transmitted bits in a repetition code (section 1.2) when the received signal is $\mathbf{r} = (1, 1, 0)$ and the channel is a binary symmetric channel with flip probability 0.1. The factors f_4 and f_5 respectively enforce the constraints that x_1 and x_2 must be identical and that x_2 and x_3 must be identical. The factors f_1, f_2, f_3 are the likelihood functions contributed by each component of $\mathbf{r}$.

A function of the factored form (26.1) can be depicted by a *factor graph*, in which the variables are depicted by circular nodes and the factors are depicted by square nodes. An edge is put between variable node n and factor node m if the function $f_m(\mathbf{x}_m)$ has any dependence on variable x_n. The factor graph for the example function (26.4) is shown in figure 26.1.

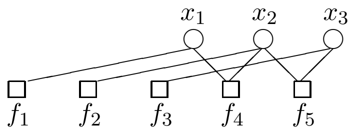

Figure 26.1. The factor graph associated with the function $P^*(\mathbf{x})$ (26.4).

The normalization problem

The first task to be solved is to compute the normalizing constant Z.

The marginalization problems

The second task to be solved is to compute the marginal function of any variable x_n, defined by

$$Z_n(x_n) = \sum_{\{x_{n'}\},\, n' \neq n} P^*(\mathbf{x}). \tag{26.5}$$

For example, if f is a function of three variables then the marginal for $n = 1$ is defined by

$$Z_1(x_1) = \sum_{x_2, x_3} f(x_1, x_2, x_3). \tag{26.6}$$

This type of summation, over 'all the $x_{n'}$ except for x_n' is so important that it can be useful to have a special notation for it – the 'not-sum' or 'summary'.

The third task to be solved is to compute the normalized marginal of any variable x_n, defined by

$$P_n(x_n) \equiv \sum_{\{x_{n'}\},\, n' \neq n} P(\mathbf{x}). \tag{26.7}$$

[We include the suffix 'n' in $P_n(x_n)$, departing from our normal practice in the rest of the book, where we would omit it.]

▷ Exercise 26.1.[1] Show that the normalized marginal is related to the marginal $Z_n(x_n)$ by

$$P_n(x_n) = \frac{Z_n(x_n)}{Z}. \tag{26.8}$$

We might also be interested in marginals over a subset of the variables, such as

$$Z_{12}(x_1, x_2) \equiv \sum_{x_3} P^*(x_1, x_2, x_3). \tag{26.9}$$

All these tasks are intractable in general. Even if every factor is a function of only three variables, the cost of computing exact solutions for Z and for the marginals is believed in general to grow exponentially with the number of variables N.

For certain functions P^*, however, the marginals can be computed efficiently by exploiting the factorization of P^*. The idea of how this efficiency arises is well illustrated by the message-passing examples of Chapter 16. The sum–product algorithm that we now review is a generalization of message-passing rule-set B (p.242). As was the case there, the sum–product algorithm is only valid if the graph is tree-like.

▶ 26.2 The sum–product algorithm

Notation

We identify the set of variables that the mth factor depends on, $\mathbf{x}_m$, by the set of their indices $\mathcal{N}(m)$. For our example function (26.4), the sets are $\mathcal{N}(1) = \{1\}$ (since f_1 is a function of x_1 alone), $\mathcal{N}(2) = \{2\}$, $\mathcal{N}(3) = \{3\}$, $\mathcal{N}(4) = \{1, 2\}$, and $\mathcal{N}(5) = \{2, 3\}$. Similarly we define the set of factors in which variable n participates, by $\mathcal{M}(n)$. We denote a set $\mathcal{N}(m)$ with variable n excluded by $\mathcal{N}(m)\backslash n$. We introduce the shorthand $\mathbf{x}_m\backslash n$ or $\mathbf{x}_{m\backslash n}$ to denote the set of variables in $\mathbf{x}_m$ with x_n excluded, i.e.,

$$\mathbf{x}_m\backslash n \equiv \{x_{n'} : n' \in \mathcal{N}(m)\backslash n\}. \tag{26.10}$$

The sum–product algorithm will involve messages of two types passing along the edges in the factor graph: messages $q_{n\to m}$ from variable nodes to factor nodes, and messages $r_{m\to n}$ from factor nodes to variable nodes. A message (of either type, q or r) that is sent along an edge connecting factor f_m to variable x_n is always a function of the variable x_n.

Here are the two rules for the updating of the two sets of messages.

From variable to factor:

$$q_{n\to m}(x_n) = \prod_{m'\in\mathcal{M}(n)\backslash m} r_{m'\to n}(x_n). \tag{26.11}$$

From factor to variable:

$$r_{m\to n}(x_n) = \sum_{\mathbf{x}_m\backslash n}\left(f_m(\mathbf{x}_m)\prod_{n'\in\mathcal{N}(m)\backslash n} q_{n'\to m}(x_{n'})\right). \tag{26.12}$$

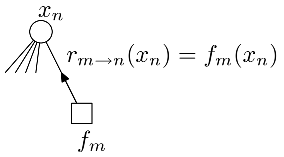

Figure 26.2. A factor node that is a leaf node perpetually sends the message $r_{m\to n}(x_n) = f_m(x_n)$ to its one neighbour x_n.

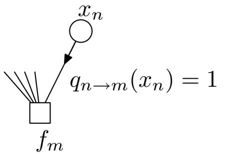

Figure 26.3. A variable node that is a leaf node perpetually sends the message $q_{n\to m}(x_n) = 1$.

How these rules apply to leaves in the factor graph

A node that has only one edge connecting it to another node is called a leaf node.

Some factor nodes in the graph may be connected to only one variable node, in which case the set $\mathcal{N}(m)\backslash n$ of variables appearing in the factor message update (26.12) is an empty set, and the product of functions $\prod_{n'\in\mathcal{N}(m)\backslash n} q_{n'\to m}(x_{n'})$ is the empty product, whose value is 1. Such a factor node therefore always broadcasts to its one neighbour x_n the message $r_{m\to n}(x_n) = f_m(x_n)$.

Similarly, there may be variable nodes that are connected to only one factor node, so the set $\mathcal{M}(n)\backslash m$ in (26.11) is empty. These nodes perpetually broadcast the message $q_{n\to m}(x_n) = 1$.

Starting and finishing, method 1

The algorithm can be initialized in two ways. If the graph is tree-like then it must have nodes that are leaves. These leaf nodes can broadcast their messages to their respective neighbours from the start.

$$\text{For all leaf variable nodes } n: \quad q_{n\to m}(x_n) = 1 \tag{26.13}$$

$$\text{For all leaf factor nodes } m: \quad r_{m\to n}(x_n) = f_m(x_n). \tag{26.14}$$

We can then adopt the procedure used in Chapter 16's message-passing rule-set B (p.242): a message is created in accordance with the rules (26.11, 26.12) only if all the messages on which it depends are present. For example, in figure 26.4, the message from x_1 to f_1 will be sent only when the message from f_4 to x_1 has been received; and the message from x_2 to f_2, $q_{2\to 2}$, can be sent only when the messages $r_{4\to 2}$ and $r_{5\to 2}$ have both been received.

Messages will thus flow through the tree, one in each direction along every edge, and after a number of steps equal to the diameter of the graph, every message will have been created.

The answers we require can then be read out. The marginal function of x_n is obtained by multiplying all the incoming messages at that node.

$$Z_n(x_n) = \prod_{m\in\mathcal{M}(n)} r_{m\to n}(x_n). \tag{26.15}$$

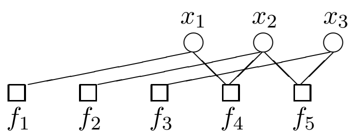

Figure 26.4. Our model factor graph for the function $P^*(\mathbf{x})$ (26.4).

The normalizing constant Z can be obtained by summing any marginal function, $Z = \sum_{x_n} Z_n(x_n)$, and the normalized marginals obtained from

$$P_n(x_n) = \frac{Z_n(x_n)}{Z}. \tag{26.16}$$

▷ Exercise 26.2.$^{[2]}$ Apply the sum–product algorithm to the function defined in equation (26.4) and figure 26.1. Check that the normalized marginals are consistent with what you know about the repetition code R_3.

Exercise 26.3.$^{[3]}$ Prove that the sum–product algorithm correctly computes the marginal functions $Z_n(x_n)$ if the graph is tree-like.

Exercise 26.4.$^{[3]}$ Describe how to use the messages computed by the sum–product algorithm to obtain more complicated marginal functions in a tree-like graph, for example $Z_{1,2}(x_1, x_2)$, for two variables x_1 and x_2 that are connected to one common factor node.

Starting and finishing, method 2

Alternatively, the algorithm can be initialized by setting *all* the initial messages from variables to 1:

$$\text{for all } n,\, m: \; q_{n\to m}(x_n) = 1, \tag{26.17}$$

then proceeding with the factor message update rule (26.12), alternating with the variable message update rule (26.11). Compared with method 1, this lazy initialization method leads to a load of wasted computations, whose results are gradually flushed out by the correct answers computed by method 1.

After a number of iterations equal to the diameter of the factor graph, the algorithm will converge to a set of messages satisfying the sum–product relationships (26.11, 26.12).

Exercise 26.5.$^{[2]}$ Apply this second version of the sum–product algorithm to the function defined in equation (26.4) and figure 26.1.

The reason for introducing this lazy method is that (unlike method 1) it can be applied to graphs that are not tree-like. When the sum–product algorithm is run on a graph with cycles, the algorithm does not necessarily converge, and certainly does not in general compute the correct marginal functions; but it is nevertheless an algorithm of great practical importance, especially in the decoding of sparse-graph codes.

Sum–product algorithm with on-the-fly normalization

If we are interested in only the *normalized* marginals, then another version of the sum–product algorithm may be useful. The factor-to-variable messages $r_{m\to n}$ are computed in just the same way (26.12), but the variable-to-factor messages are normalized thus:

$$q_{n\to m}(x_n) = \alpha_{nm} \prod_{m' \in \mathcal{M}(n)\backslash m} r_{m'\to n}(x_n) \tag{26.18}$$

where α_{nm} is a scalar chosen such that

$$\sum_{x_n} q_{n\to m}(x_n) = 1. \tag{26.19}$$

Exercise 26.6.[2] Apply this normalized version of the sum–product algorithm to the function defined in equation (26.4) and figure 26.1.

A factorization view of the sum–product algorithm

One way to view the sum–product algorithm is that it reexpresses the original factored function, the product of M factors $P^*(\mathbf{x}) = \prod_{m=1}^{M} f_m(\mathbf{x}_m)$, as another factored function which is the product of $M+N$ factors,

$$P^*(\mathbf{x}) = \prod_{m=1}^{M} \phi_m(\mathbf{x}_m) \prod_{n=1}^{N} \psi_n(x_n). \tag{26.20}$$

Each factor ϕ_m is associated with a factor node m, and each factor $\psi_n(x_n)$ is associated with a variable node. Initially $\phi_m(\mathbf{x}_m) = f_m(\mathbf{x}_m)$ and $\psi_n(x_n) = 1$.

Each time a factor-to-variable message $r_{m\to n}(x_n)$ is sent, the factorization is updated thus:

$$\psi_n(x_n) = \prod_{m \in \mathcal{M}(n)} r_{m\to n}(x_n) \tag{26.21}$$

$$\phi_m(\mathbf{x}_m) = \frac{f(\mathbf{x}_m)}{\prod_{n \in \mathcal{N}(m)} r_{m\to n}(x_n)}. \tag{26.22}$$

And each message can be computed in terms of ϕ and ψ using

$$r_{m\to n}(x_n) = \sum_{\mathbf{x}_m \backslash n} \left(\phi_m(\mathbf{x}_m) \prod_{n' \in \mathcal{N}(m)} \psi_{n'}(x_{n'}) \right) \tag{26.23}$$

which differs from the assignment (26.12) in that the product is over all $n' \in \mathcal{N}(m)$.

Exercise 26.7.[2] Confirm that the update rules (26.21–26.23) are equivalent to the sum–product rules (26.11–26.12). So $\psi_n(x_n)$ eventually becomes the marginal $Z_n(x_n)$.

This factorization viewpoint applies whether or not the graph is tree-like.

Computational tricks

On-the-fly normalization is a good idea from a computational point of view because if P^* is a product of many factors, its values are likely to be very large or very small.

Another useful computational trick involves passing the logarithms of the messages q and r instead of q and r themselves; the computations of the products in the algorithm (26.11, 26.12) are then replaced by simpler additions. The summations in (26.12) of course become more difficult: to carry them out and return the logarithm, we need to compute softmax functions like

$$l = \ln(e^{l_1} + e^{l_2} + e^{l_3}). \tag{26.24}$$

But this computation can be done efficiently using look-up tables along with the observation that the value of the answer l is typically just a little larger than $\max_i l_i$. If we store in look-up tables values of the function

$$\ln(1 + e^{\delta}) \tag{26.25}$$

(for negative δ) then l can be computed exactly in a number of look-ups and additions scaling as the number of terms in the sum. If look-ups and sorting operations are cheaper than `exp()` then this approach costs less than the direct evaluation (26.24). The number of operations can be further reduced by omitting negligible contributions from the smallest of the $\{l_i\}$.

A third computational trick applicable to certain error-correcting codes is to pass not the messages but the Fourier transforms of the messages. This again makes the computations of the factor-to-variable messages quicker. A simple example of this Fourier transform trick is given in Chapter 47 at equation (47.9).

▶ 26.3 The min–sum algorithm

The sum–product algorithm solves the problem of finding the marginal function of a given product $P^*(\mathbf{x})$. This is analogous to solving the bitwise decoding problem of section 25.1. And just as there were other decoding problems (for example, the codeword decoding problem), we can define other tasks involving $P^*(\mathbf{x})$ that can be solved by modifications of the sum–product algorithm. For example, consider this task, analogous to the codeword decoding problem:

The maximization problem. Find the setting of $\mathbf{x}$ that maximizes the product $P^*(\mathbf{x})$.

This problem can be solved by replacing the two operations add and multiply everywhere they appear in the sum–product algorithm by another pair of operations that satisfy the distributive law, namely max and multiply. If we replace summation $(+, \sum)$ by maximization, we notice that the quantity formerly known as the normalizing constant,

$$Z = \sum_{\mathbf{x}} P^*(\mathbf{x}), \tag{26.26}$$

becomes $\max_{\mathbf{x}} P^*(\mathbf{x})$.

Thus the sum–product algorithm can be turned into a max–product algorithm that computes $\max_{\mathbf{x}} P^*(\mathbf{x})$, and from which the solution of the maximization problem can be deduced. Each 'marginal' $Z_n(x_n)$ then lists the maximum value that $P^*(\mathbf{x})$ can attain for each value of x_n.

In practice, the max–product algorithm is most often carried out in the negative log likelihood domain, where max and product become min and sum. The min–sum algorithm is also known as the Viterbi algorithm.

▶ 26.4 The junction tree algorithm

What should one do when the factor graph one is interested in is not a tree?

There are several options, and they divide into exact methods and approximate methods. The most widely used exact method for handling marginalization on graphs with cycles is called the junction tree algorithm. This algorithm works by agglomerating variables together until the agglomerated graph has no cycles. You can probably figure out the details for yourself; the complexity of the marginalization grows exponentially with the number of agglomerated variables. Read more about the junction tree algorithm in (Lauritzen, 1996; Jordan, 1998).

There are many approximate methods, and we'll visit some of them over the next few chapters – Monte Carlo methods and variational methods, to name a couple. However, the most amusing way of handling factor graphs to which the sum–product algorithm may not be applied is, as we already mentioned, to apply the sum–product algorithm! We simply compute the messages for each node in the graph, as if the graph were a tree, iterate, and cross our fingers. This so-called 'loopy' message passing has great importance in the decoding of error-correcting codes, and we'll come back to it in section 33.8 and Part VI.

Further reading

For further reading about factor graphs and the sum–product algorithm, see Kschischang *et al.* (2001), Yedidia *et al.* (2000), Yedidia *et al.* (2001a), Yedidia *et al.* (2002), Wainwright *et al.* (2003), and Forney (2001).

See also Pearl (1988). A good reference for the fundamental theory of graphical models is Lauritzen (1996). A readable introduction to Bayesian networks is given by Jensen (1996).

Interesting message-passing algorithms that have different capabilities from the sum–product algorithm include *expectation propagation* (Minka, 2001) and *survey propagation* (Braunstein *et al.*, 2003). See also section 33.8.

▶ 26.5 Exercises

▷ Exercise 26.8.[2] Express the joint probability distribution from the burglar alarm and earthquake problem (example 21.1 (p.293)) as a factor graph, and find the marginal probabilities of all the variables as each piece of information comes to Fred's attention, using the sum–product algorithm with on-the-fly normalization.