COMPSCI 688: Probabilistic Graphical Models

Lecture 9: Message Passing

Dan Sheldon

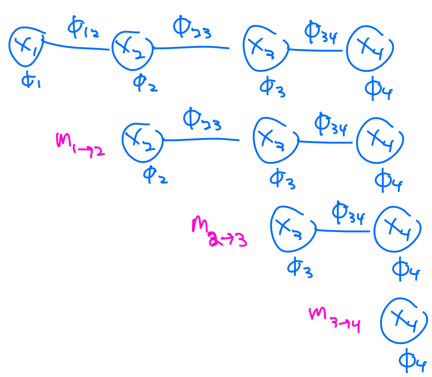
Manning College of Information and Computer Sciences
University of Massachusetts Amherst

Partially based on materials by Benjamin M. Marlin (marlin@cs.umass.edu) and Justin Domke (domke@cs.umass.edu)

Message Passing in Chains

Message Passing Derivation

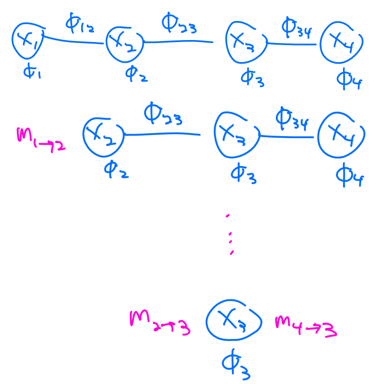
Let's go back to our chain example. Suppose we want to compute $p(x_4)$? Which variables should we eliminate, and in what order?



eliminate

x_1, x_2, x_3

What if we want to compute $p(x_3)$? Which variables should we eliminate, and in what order?



elim:
x_1, x_4, x_2 or
x_4, x_1, x_2 or
x_1, x_4, x_2

Message Passing Derivation

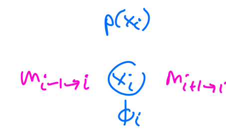
When doing "leaf-first" variable elimination to compute any marginal $p(x_i)$, there are only 6 different intermediate factors

$$m_{1\to 2}, m_{2\to 3}, m_{3\to 4}, \quad m_{4\to 3}, m_{3\to 2}, m_{2\to 1}$$

Let's call $m_{j\to i}$ the "message" from j to i.

We can compute Z by "collecting" messages at any node:

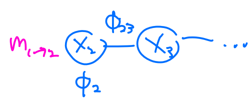
$$\boxed{Z = \sum_{x_i} \phi_i(x_i) \prod_{j\in \mathsf{nb}(i)} m_{j\to i}(x_i)}$$



The general formula for a marginal is similar, but we omit the final summation and normalize:

$$\boxed{p(x_i) = \frac{1}{Z}\phi_i(x_i) \prod_{j\in \mathsf{nb}(i)} m_{j\to i}(x_i)}$$

Message Passing Derivation



The messages satisfy recurrences, e.g.

$$m_{2\to 3}(x_3) = \sum_{x_2} m_{1\to 2}(x_2)\phi_2(x_2)\phi_{23}(x_2, x_3)$$

The message $m_{i-1\to i}(x_i)$ sums out all variables from the product of all factors "to the left" of x_i

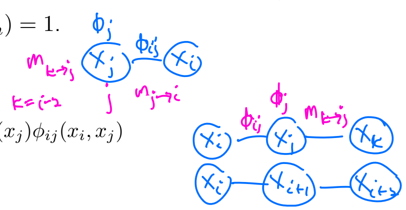
The message $m_{i+1\to i}(x_i)$ has a similar recurrence, and sums out variables/factors "to the right".

Using the recurrences, we can compute *all messages*, and therefore *all marginals* in two passes through the chain, one in each direction.

Message Passing in a Chain

$m_{0\to 1}$ $m_{1\to 2}, m_{2\to 3} \cdots m_{n-1\to n}$
$m_{2\to 1}$ $\cdots$ $m_{n\to n-1}$ $m_{n+1\to n}$

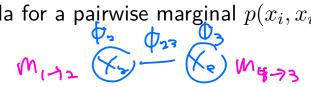
- Initialize $m_{0\to 1}(x_1) = 1$, $m_{n+1\to n}(x_n) = 1$.
- For $i = 2$ to n
  - Let $k = i - 2$, $j = i - 1$
  - Let $m_{j\to i}(x_i) = \sum_{x_j} m_{k\to j}(x_j)\phi_j(x_j)\phi_{ij}(x_i, x_j)$
- For $i = n - 1$ down to 1
  - Let $k = i + 2$, $j = i + 1$
  - Let $m_{j\to i}(x_i) = \sum_{x_j} m_{k\to j}(x_j)\phi_j(x_j)\phi_{ij}(x_i, x_j)$



- Compute each unnormalized marginal as $\hat{p}(x_i) = m_{i-1\to i}(x_i)\phi_i(x_i)m_{i+1\to i}(x_i)$
- Compute $Z = \sum_{x_i} \hat{p}(x_i)$ for any i, and normalize each marginal: $p(x_i) = \frac{1}{Z}\hat{p}(x_i)$

Pairwise Marginals

- Correct formula for a pairwise marginal $p(x_i, x_{i+1})$?



$$p(x_i, x_{i+1}) = \frac{1}{Z} m_{i-1\to i}(x_i)\,\phi_i(x_i)\,\phi_{i,i+1}(x_i, x_{i+1})\,\phi_{i+1}(x_{i+1})\,m_{i+2\to i+1}(x_{i+1})$$

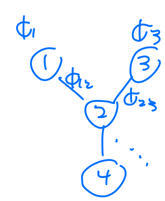
Discussion: Message Passing vs. Variable Elimination

- Variable elimination can compute marginals and Z **exponentially faster** than direct summation for nice enough graphs (e.g. chains, trees)
- Naively, to compute all single-node marginals you would have to run variable elimination n times, once per node (but this would repeat work)
- Message passing can compute all the marginals for the same cost as running variable elimination twice, so is a **factor of** $\approx n/2$ **faster** than naive variable elimination
- (Message passing is nice, but you could say variable elimination did the heavy lifting.)

Message Passing in Trees

Message Passing in Trees

A more general version of message passing works for any *tree-structured MRF*, that is, an MRF of the following form where $G = (V, E)$ is a tree:



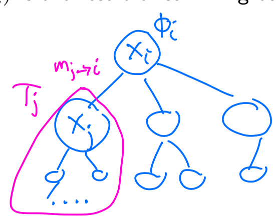
$$p(\mathbf{x}) = \prod_{i \in V} \phi_i(x_i) \prod_{(i,j)\in E} \phi_{ij}(x_i, x_j).$$

Message passing can be derived from variable elimination. Take x_i as the root and eliminate variables from leaf to root. We get

$$Z = \sum_{x_i} \phi_i(x_i) \prod_{j \in \mathsf{nb}(i)} m_{j \to i}(x_i)$$

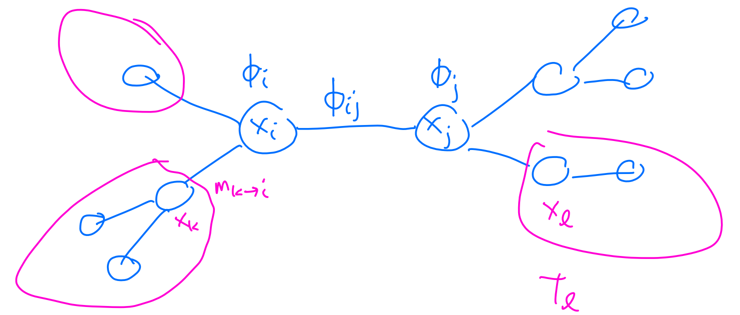
$$p(x_i) = \frac{1}{Z} \phi_i(x_i) \prod_{j \in \mathsf{nb}(i)} m_{j \to i}(x_i)$$

The "message" $m_{j \to i}(x_i)$ is the result of summing out all factors and variables in the subtree T_j rooted at x_j.



By similar reasoning, the pairwise marginal for $(i, j) \in E$ is

$$p(x_i, x_j) = \frac{1}{Z} \phi_i(x_i) \phi_{ij}(x_i, x_j) \phi_j(x_j) \prod_{k \in \mathsf{nb}(i) \setminus j} m_{k \to i}(x_i) \prod_{\ell \in \mathsf{nb}(j) \setminus i} m_{\ell \to j}(x_j)$$

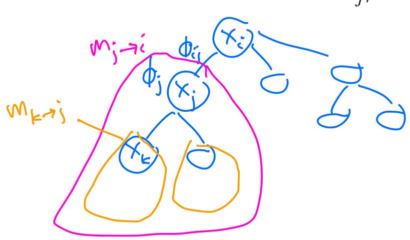


Recurrence for Messages

The messages satisfy the following recurrence

$$\boxed{m_{j \to i}(x_i) = \sum_{x_j} \phi_j(x_j) \phi_{ij}(x_i, x_j) \prod_{k \in \mathsf{nb}(j) \setminus i} m_{k \to j}(x_j)}$$

This can be understood by expanding the summation over T_j to group factors for subtrees rooted at each child of x_j, that is, for each node $k \in \mathrm{nb}(j) \setminus i$.



Message-Passing

Importantly, the message from j to i doesn't depend on which particular node is the root. There are only $2(n-1)$ total messages and we can compute them all in two passes through the tree.

Say that j is **ready to send to** i if j has received messages from all $k \in \mathrm{nb}(j) \setminus i$.

Message passing: while any node j is ready to send to i, compute $m_{j \to i}$ using recurrence from previous slide.

This algorithm is described asynchronously ("ready-to-send"), but in practice: pass messages from leaves to root of tree and back.

Message-Passing Summary

$$m_{j\to i}(x_i) = \sum_{x_j} \phi_j(x_j)\phi_{ij}(x_i,x_j) \prod_{k\in \mathsf{nb}(j)\setminus i} m_{k\to j}(x_j)$$

recurrence

$$Z = \sum_{x_i} \phi_i(x_i) \prod_{j\in \mathsf{nb}(i)} m_{j\to i}(x_i)$$

Z

$$p(x_i) = \frac{1}{Z}\phi_i(x_i) \prod_{j\in \mathsf{nb}(i)} m_{j\to i}(x_i)$$

single marginal

$$p(x_i,x_j) = \frac{1}{Z}\phi_i(x_i)\phi_{ij}(x_i,x_j)\phi_j(x_j) \prod_{k\in \mathsf{nb}(i)\setminus j} m_{k\to i}(x_i) \prod_{\ell\in \mathsf{nb}(j)\setminus i} m_{\ell\to j}(x_j) \quad (i,j)\in E$$

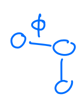
pairwise marginal

Discussion and Extensions

Discussion

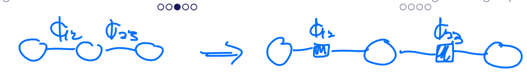
Trees - $p(x_1)$ $p(x_2)$ $p(x_3)$ .... $p(x_1,x_2)$ $p(x_2,x_3)$

- Message-passing computes *all single and pairwise marginals* at roughly 2x cost of variable elimination
- It is restricted to pairwise MRFs and trees, but can be extended in some ways
- For exactly answering *one query* in any MRF, variable elimination is faster than message passing
- For exactly answering a set of marginal queries, variable elimination usually takes at most a factor of $O(n)$ more time

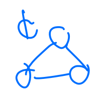


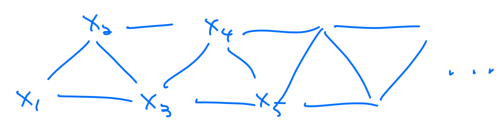
$\phi_{123}(x_1,x_2,x_3)$

Sketches of Extensions



- What if the MRF has factors on more than two variables? (keyword: *factor graphs*)

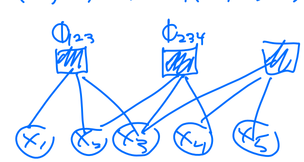




$\phi_{123}(x_1,x_2,x_3)\,\phi_{234}(x_2,x_3,x_4)$ ....

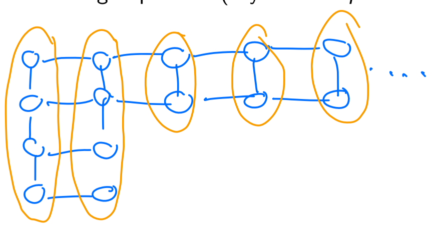
factors

variables



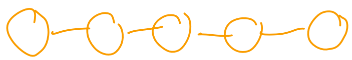
messages passed from factors to variables + vice versa

- What if the MRF is not tree-structured, i.e., G has cycles?
- **Answer 1**: group nodes (keyword: *clique trees* or *junction trees*) exact inference



tree decomposition

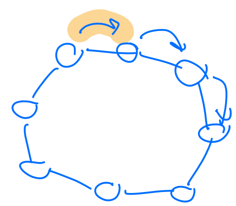
"treewidth"



- What if the MRF is not tree-structured, i.e., G has cycles?
- **Answer 2**: use message-passing as a fixed-point iteration (keyword: *loopy belief propagation*)

Init

$m_{i\to j}(x_j) \equiv 1$ for all $(i,j) \in E$



More general schemes for approx inference

— MCMC

— Variational inference

Message-Passing Implementation

Overflow/Underflow and Log-Sum-Exp

- When factor values are small or large, or with many factors, messages can underflow or overflow since they are products of many terms. A common solution is to manipulate all factors and messages in log space.
- **Example**: consider the common factor manipulation

$$A(x) = \sum_y B(x,y)C(y)$$

Let's compute $\alpha(x) = \log A(x)$ from $\beta(x,y) = \log B(x,y)$ and $\gamma(y) = \log C(y)$

- **Step 1**: multiplication of factors is addition of log-factors

$$\lambda(x,y) := \log(B(x,y)C(y)) = \beta(x,y) + \gamma(y)$$

- **Step 2**: marginalization requires exponentiation ("log-sum-exp")

$$\alpha(x) = \log\left(\sum_y \exp\lambda(x,y)\right)$$

Numerically Stable log-sum-exp

Before exponentiating, we need to be careful to shift values to avoid overflow/underflow

logsumexp$(a_1, \ldots, a_k)$:

- $c \leftarrow \max_i a_i$
- return $c + \log \sum_i \exp(a_i - c)$

See `scipy.special.logsumexp`

(Comment: log-space implementation probably not needed in HW2, probably needed in HW3.)