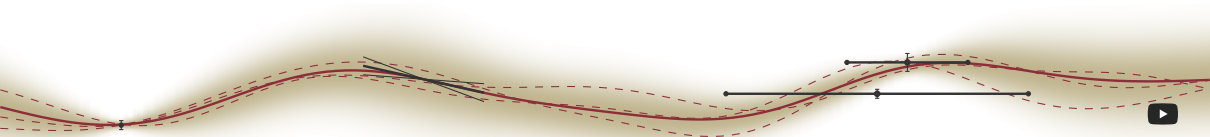
Probabilistic Machine Learning
Lecture 18
The Sum-Product Algorithm

Philipp Hennig

23 June 2020

Eberhard Karls
Universität
Tübingen

Faculty of Science
Department of Computer Science
Chair for the Methods of Machine Learning



#	date	content	Ex	#	date	content	Ex
1	20.04.	Introduction	1	14	09.06.	Generalized Linear Models	
2	21.04.	Reasoning under Uncertainty		15	15.06.	Exponential Families	8
3	27.04.	Continuous Variables	2	16	16.06.	Graphical Models	
4	28.04.	Monte Carlo		17	22.06.	Factor Graphs	9
5	04.05.	Markov Chain Monte Carlo	3	18	23.06.	The Sum-Product Algorithm	
6	05.05.	Gaussian Distributions		19	29.06.	Example: Topic Models	10
7	11.05.	Parametric Regression	4	20	30.06.	Mixture Models	
8	12.05.	Learning Representations		21	06.07.	EM	11
9	18.05.	Gaussian Processes	5	22	07.07.	Variational Inference	
10	19.05.	Understanding Kernels		23	13.07.	Topics	12
11	26.05.	Gauss-Markov Models		24	14.07.	Example: Inferring Topics	
12	25.05.	An Example for GP Regression	6	25	20.07.	Example: Kernel Topic Models	
13	08.06.	GP Classification	7	26	21.07.	Revision	

The Toolbox

Framework:

$$\int p(x_1, x_2)\, dx_2 = p(x_1) \qquad p(x_1, x_2) = p(x_1 \mid x_2) p(x_2) \qquad p(x \mid y) = \frac{p(y \mid x) p(x)}{p(y)}$$

Modelling:

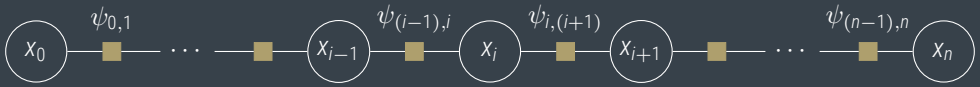
- graphical models
- Gaussian distributions
- Kernels
- Markov Chains
- Exponential Families / Conjugate Priors
- Factor Graphs & Message Passing

Computation:

- Monte Carlo
- Linear algebra / Gaussian inference
- maximum likelihood / MAP
- Laplace approximations
-

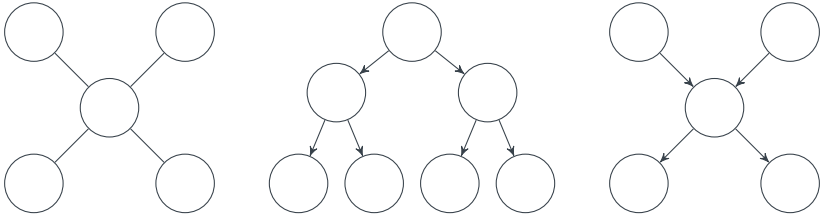
Factor Graphs

- are a tool to directly represent an entire computation in a formal language (which also includes the functions in question themselves)
- both directed and undirected graphical models can be mapped onto factor graphs.



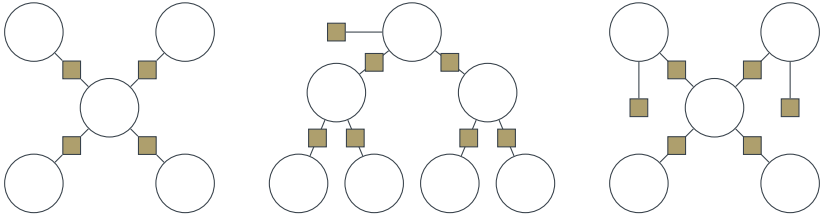
Inference on Chains

- separates into **local messages** being sent forwards and backwards along the factor graph
- both the local marginals and the *most-probable state* can be inferred in this way



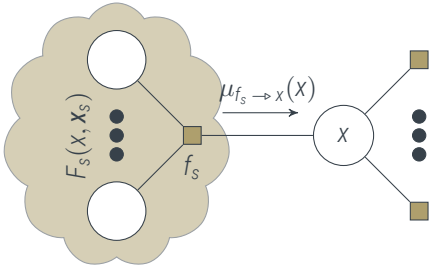
Definition (Tree)

An *undirected* graph is a **tree** if there is one, and only one, path between any pair of nodes (such graphs have no loops). A *directed* graph is a **tree** if there is only one node which has no parent (the *root*), and all other nodes have only one parent. When such graphs are transformed into undirected graphs by moralization, they remain a tree. A directed graph such that every pair of nodes is connected by one and only one path is called a **polytree**. When transformed into an undirected graph, such graphs, in general, acquire loops. But the corresponding factor graph is still a tree.



Definition (Tree)

An *undirected* graph is a **tree** if there is one, and only one, path between any pair of nodes (such graphs have no loops). A *directed* graph is a **tree** if there is only one node which has no parent (the *root*), and all other nodes have only one parent. When such graphs are transformed into undirected graphs by moralization, they remain a tree. A directed graph such that every pair of nodes is connected by one and only one path is called a **polytree**. When transformed into an undirected graph, such graphs, in general, acquire loops. But the corresponding factor graph is still a tree.



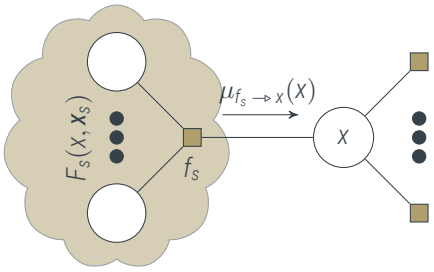
- Consider a tree-structured factor graph over $\boldsymbol{x} = [x_1, \ldots, x_n]$ (if instead you have an undirected tree or directed polytree, transform it first).
- Again, w.l.o.g. assume discrete variables for simplicity (for continuous, replace sums by integrals).
- Pick any variable $x \in \boldsymbol{x}$. **Because the graph is a tree**, we can write

$$p(\boldsymbol{x}) = \prod_{s \in \mathrm{ne}(x)} F_s(x, \boldsymbol{x}_s)$$

  where $\mathrm{ne}(x)$ are the **neighbors** of x, and F_s is the **sub-graph** of nodes $\boldsymbol{x}_s$ other than x itself that are connected to neighbor s (which is itself a tree!).
- Consider the **marginal** distribution $p(x) = \sum_{\boldsymbol{x} \setminus x} p(\boldsymbol{x})$

$$a_1 \cdot b_1 + a_1 \cdot b_2 + a_2 \cdot b_1 + a_2 \cdot b_2 = (a_1 + a_2) \cdot (b_1 + b_2)$$

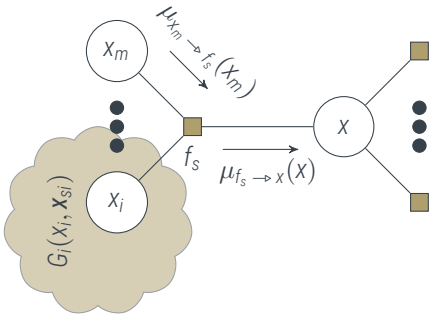
$$\sum_i \prod_j f_{ij} = \prod_j \sum_i f_{ij}$$



$$p(x) = \sum_{\boldsymbol{x} \setminus x} \prod_{s \in \mathrm{ne}(x)} F_s(x, \boldsymbol{x}_s) = \prod_{s \in \mathrm{ne}(x)} \underbrace{\left(\sum_{\boldsymbol{x}_s} F(x, \boldsymbol{x}_s) \right)}_{=: \mu_{f_s \to x}(x)}$$

$$= \prod_{s \in \mathrm{ne}(x)} \mu_{f_s \to x}(x)$$

The marginal $p(x)$ is a product of **incoming messages** $\mu_{f_s \to x}$ from the factors connected to x.



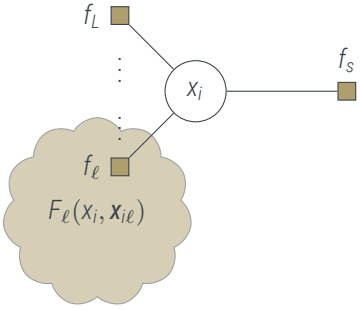
- consider the sub-graph $F_s(x, \boldsymbol{x}_s)$ and factorize **that** sub-graph into further (tree-structured) sub-graphs

$$F_s(x, \boldsymbol{x}_s) = f_s(x, x_1, \dots, x_m) G_1(x_1, \boldsymbol{x}_{s1}) \cdots G_m(x_m, \boldsymbol{x}_{sm})$$

- then we can write

$$\begin{aligned}\mu_{f_s \to x}(x) &= \sum_{x_1,\dots,x_m} f_s(x, x_1, \dots, x_m) \prod_{i \in \text{ne}(f_s) \setminus x} \underbrace{\left(\sum_{\boldsymbol{x}_{si}} G_i(x_i, \boldsymbol{x}_{si}) \right)}_{\mu_{x_i \to f_s}(x_i)} \\ &= \sum_{x_1,\dots,x_m} f_s(x, x_1, \dots, x_m) \prod_{i \in \text{ne}(f_s) \setminus x} \mu_{x_i \to f_s}(x_i)\end{aligned}$$

To compute the factor-to-variable message $\mu_{f_s \to x}(x)$, **sum** over the **product** of the factor and remaining sub-graph-sums. The latter are themselves **messages** from the variables connected to f_s.



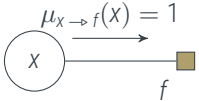
$$G_i(x_i, \boldsymbol{x}_{si}) = \prod_{\ell \in \mathrm{ne}(x_i) \setminus f_s} F_\ell(x_i, \boldsymbol{x}_{i\ell})$$

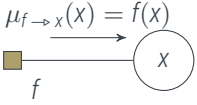
$$\mu_{x_i \to f_s}(x_i) = \sum_{\boldsymbol{x}_{si}} G_i(x_i, \boldsymbol{x}_{si}) = \sum_{\boldsymbol{x}_{si}} \left(\prod_{\ell \in \mathrm{ne}(x_i) \setminus f_s} F_\ell(x_i, \boldsymbol{x}_{i\ell}) \right)$$

$$= \prod_{\ell \in \mathrm{ne}(x_i) \setminus f_s} \left(\sum_{\boldsymbol{x}_{i\ell}} F_\ell(x_i, \boldsymbol{x}_{i\ell}) \right)$$

$$= \prod_{\ell \in \mathrm{ne}(x_i) \setminus f_s} \mu_{f_\ell \to x_i}(x_i)$$

To compute the variable-to-factor message $\mu_{x_i \to f_s}(x_i)$, take the **product** of all incoming factor-to-variable messages. Repeat recursively, until reaching a **leaf** node.





$$\mu_{x \to f}(x) = \prod_{\varnothing} \sum_{\varnothing} := 1$$

$$\mu_{f \to x}(x) = \sum_{\varnothing} f(x, \varnothing) \prod_{\varnothing} := f(x)$$

To initiate the messages at leaves of the graph, define them to be unit for variable leaves and identities for factor leaves.

To compute the marginal $p(x)$, treat it as the root of the tree, and do:

- start at leaf nodes.
  - if leaf is factor $f(x)$, initialize $\mu_{f \to x}(x) = f(x)$
  - if leaf is variable x, initialize $\mu_{x \to f}(x) = 1$
- pass messages from the leaves towards the root x:

$$\mu_{f_\ell \to x_j} = \sum_{\boldsymbol{x}_{\ell j}} f_\ell(x_j, \boldsymbol{x}_{\ell j}) \prod_{i \in \{\ell j\} = \mathrm{ne}(f_\ell) \setminus x_j} \mu_{x_i \to f_\ell}(x_i) \qquad \mu_{x_j \to f_\ell}(x_j) = \prod_{i \in \mathrm{ne}(x_j) \setminus f_\ell} \mu_{f_i \to x_j}(x_j)$$

- at the root x, take product of all incoming messages (and normalize).

The Sum-Product Algorithm

Summary

To compute the marginals $p(x)$ of **all** variables, choose *any* x_i as the root. Then,

- start at leaf nodes.
  - if leaf is factor $f(x)$, initialize $\mu_{f \to x}(x) = f(x)$
  - if leaf is variable x, initialize $\mu_{x \to f}(x) = 1$
- pass messages from leaves towards root:

$$\mu_{f_\ell \to x_j} = \sum_{\boldsymbol{x}_{\ell j}} f_\ell(x_j, \boldsymbol{x}_{\ell j}) \prod_{i \in \{\ell j\} = \mathrm{ne}(f_\ell) \setminus x_i} \mu_{x_i \to f_\ell}(x_i) \qquad \mu_{x_j \to f_\ell}(x_j) = \prod_{i \in \mathrm{ne}(x_j) \setminus f_\ell} \mu_{f_i \to x_j}(x_j)$$

- once root has messages from all neighbors, pass messages **from** to root **towards the leaves**.
- once all nodes have received messages from all their neighbors, take product of all incoming messages at all variables (and normalize).

Inference on the marginal of all variables in a tree-structured factor-graph is **linear** in graph size.

- The two types of messages can be combined, phrasing the algorithm as message passing between factor nodes only:

$$\mu_{f_\ell \to x_j} = \sum_{\boldsymbol{x}_{\ell j}} f_\ell(x_j, \boldsymbol{x}_{\ell j}) \prod_{i \in \mathrm{ne}(f_\ell) \setminus x_j} \mu_{x_i \to f_\ell}(x_i)$$

$$\mu_{x_j \to f_\ell}(x_j) = \prod_{i \in \mathrm{ne}(x_j) \setminus f_\ell} \mu_{f_i \to x_j}(x_j)$$

$$m_{f_\ell \to f_j}(\boldsymbol{x}_j) = \sum_{\boldsymbol{x}_\ell \setminus (\boldsymbol{x}_\ell \cap \boldsymbol{x}_j)} f_\ell(x_j, \boldsymbol{x}_\ell) \prod_{i \in \mathrm{ne}(f_\ell) \setminus \mathrm{ne}(f_j)} m_{f_i \to f_j}(x_\ell)$$

- If one or more nodes $\boldsymbol{x}^o$ in the graph are **observed** ($\boldsymbol{x}^o = \hat{\boldsymbol{x}}^o$), just introduce factors $f(x_i^o) = \delta(x_i^o - \hat{x}_i^o)$ into the graph.
- This amounts to "clamping" the variables to their observed value
- Say $\boldsymbol{x} := [\boldsymbol{x}^o, \boldsymbol{x}^h]$. Because $p(\boldsymbol{x}^o, \boldsymbol{x}^h) \propto p(\boldsymbol{x}^h \mid \boldsymbol{x}^o)$, the sum-product algorithm can thus be used to compute *posterior* marginal distributions over the hidden variables $\boldsymbol{x}^h$.

Some Notes

Inference by message passing in graphs

- There is a generalization from trees to general graphs, known as the **junction tree algorithm**. The principal idea is to **join** sets of variables in the graph into larger maximal cliques until the resulting graph is a tree. The exact process, however, requires care to ensure that every clique that is a sub-set of another clique ends up in that clique. The resulting algorithm (like the sum-product algorithm) has complexity exponential in the dimensionality of the largest variable in the graph, and linear in the size of tree.

The computational cost of probabilistic inference on the marginal of a variable in a joint distribution is exponential in the dimensionality of the maximal clique of the juntion tree, and linear in the size of the junction tree. The junction tree algorithm is **exact** for any graph (it produces correct martginals), and **efficient** in the sense that, given a graph, there does not in general (i.e. without using properties of the functions instead of the graph) exist a more efficient algorithm.

What if we don't care about the marginal posteriors,
but about the **joint** distribution?

In general, it's shape can be very complex, and exponentially hard to track (in the number of variables). But remember from lecture 1 that *storing* the **maximum** of the distribution has linear complexity (just write it down!).

How about **computing** that maximum?

The Max-Product / Max-Sum Algorithm

Finding Most Probable Configurations

[Exposition from Bishop, PRML, 2006]

- What if, instead of marginals $p(x_i)$ we want the jointly **most probable** state $x^{\text{max}} = \arg\max_{\boldsymbol{x}} p(\boldsymbol{x})$?
- note that $\arg\max_{\boldsymbol{x}} p(\boldsymbol{x}) \neq \prod \arg\max_{x_i} p(x_i)$:

		0.6	0.4
		$x_2 = 0$	$x_2 = 1$
0.7	$x_1 = 0$	0.3	**0.4**
0.3	$x_1 = 1$	0.3	0.0

- but $\max(ab, ac) = a\max(b, c)$ and $\max(a + b, a + c) = a + \max(b, c)$! Also (cf. earlier lectures)

$$\log\left(\max_{\boldsymbol{x}} p(\boldsymbol{x})\right) = \max_{\boldsymbol{x}} \log p(\boldsymbol{x})$$

Thus, we can compute the most probable state x^{max} by taking the sum-product algorithm and replacing all summations with maximizations (the **max-product** algorithm). We can further replace all products of p with sums of $\log p$ (the **max-sum** algorithm). The only complication is that, if we also want to know the arg max, we have to track it separately, using an additional data structure.

The Max-Product Algorithm

Summary

To compute $\boldsymbol{x}^{\max}$, choose *any* x_i as the root. Then,

- start at leaf nodes.
  - if leaf is factor $f(x)$, initialize $\mu_{f \to x}(x) = f(x)$
  - if leaf is variable x, initialize $\mu_{x \to f}(x) = 1$
- pass messages from leaves towards root:

$$\mu_{f_\ell \to x_j}(x_j) = \max_{\boldsymbol{x}_{\ell j}} f_\ell(x_j, \boldsymbol{x}_{\ell j}) \prod_{i \in \{\ell j\} = \mathrm{ne}(f_\ell) \setminus x_j} \mu_{x_i \to f_\ell}(x_i) \qquad \mu_{x_j \to f_\ell}(x_j) = \prod_{i \in \mathrm{ne}(x_j) \setminus f_\ell} \mu_{f_i \to x_j}(x_j)$$

- additionally track indicator for **identity** of maximum (nb: This is a function of x_j!)

$$\phi(x_j) = \arg\max_{\boldsymbol{x}_{\ell j}} f_\ell(x_j, \boldsymbol{x}_{\ell j}) \prod_{i \in \mathrm{ne}(f_\ell) \setminus x_j} \mu_{x_i \to f_\ell}(x_i)$$

- once root has messages from all neighbors, pass messages **from** to root **towards the leaves**. At each factor node, set $\boldsymbol{x}_{\ell j}^{\max} = \phi(x_j)$ (this is known as **backtracking**).

The Max-Sum Algorithm

Summary

To compute $\boldsymbol{x}^{\max}$, choose *any* x_i as the root. Then,

- start at leaf nodes.
  - if leaf is factor $f(x)$, initialize $\mu_{f \to x}(x) = \log f(x)$
  - if leaf is variable x, initialize $\mu_{x \to f}(x) = 0$
- pass messages from leaves towards root:

$$\mu_{f_\ell \to x_j}(x_j) = \max_{\boldsymbol{x}_{\ell j}} \log f_\ell(x_j, \boldsymbol{x}_{\ell j}) + \sum_{i \in \{\ell j\} = \mathrm{ne}(f_\ell) \setminus x_j} \mu_{x_i \to f_\ell}(x_i) \qquad \mu_{x_j \to f_\ell}(x_j) = \sum_{i \in \mathrm{ne}(x_j) \setminus f_\ell} \mu_{f_i \to x_j}(x_j)$$

- additionally track indicator for **identity** of maximum (nb: This is a function of x_j!)

$$\phi(x_j) = \arg\max_{x_{\ell j}} \log f_\ell(x_j, \boldsymbol{x}_{\ell j}) + \sum_{i \in \mathrm{ne}(f_\ell) \setminus x_j} \mu_{x_i \to f_\ell}(x_i)$$

- once root has messages from all neighbors, pass messages **from** to root **towards the leaves**. At each factor node, set $\boldsymbol{x}_{\ell j}^{\max} = \phi(x_j)$ (this is known as **backtracking**).

- Max-Sum is a case of **dynamic programming** (recursive simplification of optimization using problem structure). The equation

$$\mu_{f_\ell \to f_j} = \max_{\mathbf{x}_\ell \setminus (\mathbf{x}_\ell \cap \mathbf{x}_j)} \left(\log f_\ell(x_\ell) + \sum_{i \in \mathrm{ne}(f_\ell) \setminus \mathrm{ne}(f_j)} \mu_{f_i \to f_j}(\mathbf{x}_j) \right)$$

  defines a **Hamilton-Jacobi-Bellman equation**

Summary:

- Factor graphs provide graphical representation of joint probability distributions that is particularly conducive to automated inference
- In factor graphs that are *trees*, all **marginals** can be computed in time **linear** in the graph size by **passing messages** along the edges of the graph using the **sum-product** algorithm.
- Computation of each local marginal is exponential in the dimensionality of the node. Thus, in general, the cost of inference is exponential in clique-size, linear in clique-number.
- An analogous algorithm, the **max-sum** algorithm, can be used to find the joint most probable state, also in linear time.
- Both algorithms fundamentally rest on the distributive properties

$$a(b+c) = ab + ac \qquad \max(ab, ac) = a \cdot \max(b, c)$$

Message passing provides the general framework for managing computational complexity in probabilistic generative models as far as it is caused by **conditional independence**. It does not, however, address complexity arising from the algebraic form of continous probability distributions. We already saw that **exponential families** address this latter issue. But not every distribution is an exponential family. A main theme for the remainder will be how to project complicated joint distributions onto factor graphs of exponential families.