Yet another introduction to probabilistic (graphical) models, or What de fudge are PGMs?

Herman Kamper

2023-01

Strategy

Instead of directly jumping into the details (as most courses do), I first want to try and give the bigger picture. But to do this it is helpful to know how probabilistic graphical models (PGMs) are represented. So I first cover representation, then give the bigger picture, and only then get into the details.

Representation

Barber (2020, Sec. 4.1):

> Whilst not a strict separation, GMs tend to fall into two broad classes—those useful in modelling, and those useful in representing inference algorithms. For modelling, belief networks, Markov networks, chain graphs and influence diagrams are some of the most popular. For inference one typically "compiles" a model into a suitable GM for which an algorithm can be readily applied. Such inference GMs include factor graphs and junction trees.

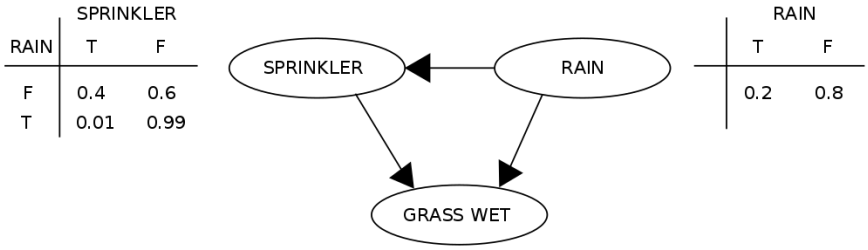
This section reviews PGM representations that are useful for modelling. Representations useful for inference (specifically factor and cluster graphs) are covered later.

Directed PGMs: Bayesian networks

Different names for the same thing, fortunately all with the same abbreviation (BN):

- Belief network
- Bayes net(work)
- Bayesian network

A directed graph:[1]



RAIN	SPRINKLER T	SPRINKLER F
F	0.4	0.6
T	0.01	0.99

RAIN T	RAIN F
0.2	0.8

SPRINKLER	RAIN	GRASS WET T	GRASS WET F
F	F	0.0	1.0
F	T	0.8	0.2
T	F	0.9	0.1
T	T	0.99	0.01

The above Bayesian network encodes the joint:

$$P(g, s, r) = P(g|s, r)P(s|r)P(r)$$

Storing the conditional probability tables are often more efficient than storing the full joint

(However, in the example PGM this isn't actually true. But if sprinkler was independent of rain, it would be more efficient.)

[1] Figure from Wikipedia.

In general the joint probability density for a Bayesian network:

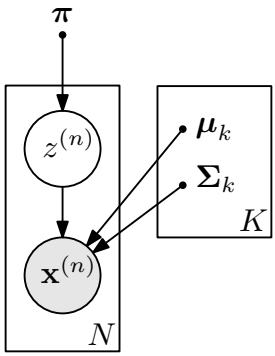
$$p(x_1, x_2, \ldots, x_D) = \prod_{d=1}^{D} p\left(x_d | \mathrm{pa}(x_d)\right)$$

Graphically:

- Random variables are round nodes.
- Observed variables are shaded.
- Repetitions are indicated using plate notation.
- Parameters (or hyper-parameters) that are not considered random variables are dots (sometimes).

Bayesian networks are useful for graphically reading off (conditional) independencies (Barber 2020, Def. 3.3).

Example: Gaussian mixture model (GMM)

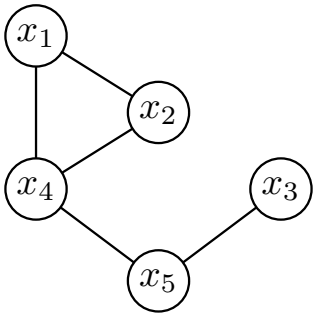


Undirected PGMs: Markov random fields

Different names for the same thing:

- Markov random field
- Markov network

An undirected graph:



Each edge represents a dependency: x_1 depends on x_2 and x_4; x_2 depends on x_1 and x_4; x_4 depends on x_1, x_2, and x_5; x_5 depends on x_4 and x_3; and x_3 depends on x_5.

The above Markov random field encodes the joint density:

$$p(x_1, x_2, x_3, x_4, x_5) = \frac{1}{Z}\phi_1(x_1, x_2, x_4)\phi_2(x_4, x_5)\phi_3(x_3, x_5)$$

where Z is a normalisation constant.

The $\phi(x_1, x_2, \ldots, x_D)$ functions:

- Are called *factors* or *potentials*.
- Are non-negative.
- A probability density function is a special case of a factor, one that normalises to 1. (They don't normalise to 1 in general.)

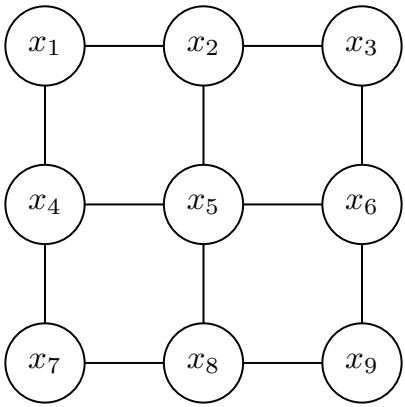
In general the joint probability density for a Markov random field:

$$p(x_1, x_2, \ldots, x_D) = \frac{1}{Z} \prod_{c=1}^{C} \phi_c(\mathcal{X}_c)$$

where the product is over the (maximal) cliques in the graph, $\mathcal{X}_c$ is the subset of variables in clique c, and Z is a normalisation constant.

Example: Ising model

Example from (Barber 2020, Sec. 4.2.5). You have a grid of "mini-magnets", each with a binary value $x_i \in \{-1, +1\}$ indicating its direction. Neighbouring magnets want to have the same orientation.



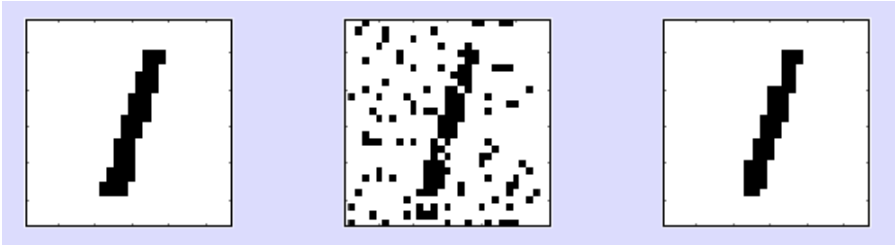
$$p(x_1, x_2, \ldots, x_9) = \frac{1}{Z} \prod_{i \sim j} \phi_{i,j}(x_i, x_j)$$

where $i \sim j$ are indices where i and j are neighbours.

$$\phi_{i,j}(x_i, x_j) = \exp\left\{-\frac{1}{2\tau}(x_i - x_j)^2\right\}$$

where τ is a temperature parameter.

Can be used for simple image restoration:[2]



[2]Figure from (Barber 2020).

Bigger picture

1. Representation

How is a model drawn, written, mathematically expressed, or coded? Some representations are useful for modelling (main ones are described above) while others are useful for inference (below).

2. Inference

Given that the grass is wet, what is the probability that it rained $P(r|g = \texttt{wet})$? We can assume we know the values in the tables (i.e. the parameters are known).

Inference is what we do when we use a model to answer questions.

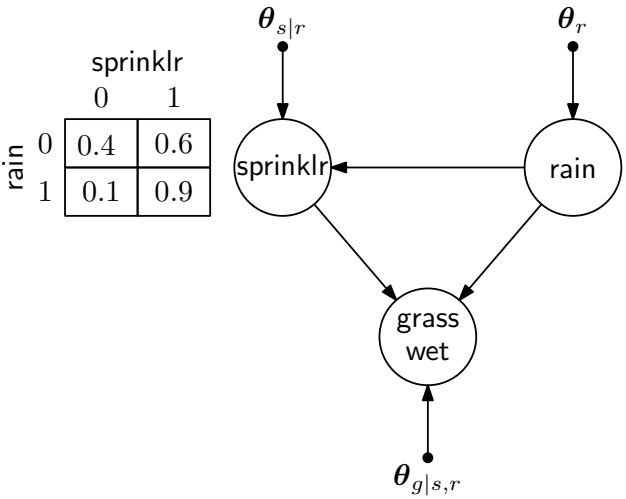
More formally: If we have a probabilistic model for which we know all the parameters, then inference (reasoning) is performed by setting some of the variables to observed states, and then computing probabilities of interest conditioned on this evidence (Barber 2020, Sec. 2.1).

3. Learning

We have a model with parameters $\boldsymbol{\theta}$. Given that we have observed some data $\mathcal{D}$, what is the $\boldsymbol{\theta}$ that best explains the data?

Rain example: Now we do not know the parameters in the tables, and we need to learn these from previously observed data $(g^{(n)}, s^{(n)}, r^{(n)})$.

Latent variables: Very often in learning we also have latent (hidden) variables that are not observed directly. E.g. in a GMM we can observe $\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots \mathbf{x}^{(N)}\}$, but we don't have direct access to $\{z^{(1)}, z^{(2)}, \dots z^{(N)}\}$.



Strategy

Many PGM courses systematically go through (1), (2), (3) in sequence, but this is tricky since some representations are coupled with specific inference methods and (as we shall see very soon) inference and learning are often closely tied.

So my strategy will be to jump around, with the aim of starting with concepts that are more familiar and then moving to the less familiar. I do find it helpful to sometimes just take a step back and ask myself whether I am busy with (1), (2) or (3).

To illustrate things in the remainder of this note, I will mainly use Bayesian networks.

Learning

We want to learn the parameters $\boldsymbol{\theta}$ of a model so that it best describes observed data $\mathcal{D}$ (Barber 2020, Sec. 8.6).

- Supervised learning: $\mathcal{D} = \left\{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \ldots, (\mathbf{x}^{(N)}, y^{(N)})\right\}$
- Unsupervised learning: $\mathcal{D} = \{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(N)}\}$

There are several ways of learning $\boldsymbol{\theta}$.

Maximum likelihood estimation

$$\boxed{\hat{\boldsymbol{\theta}}_{\text{MLE}} = \arg\max_{\boldsymbol{\theta}} p(\mathcal{D}|\boldsymbol{\theta}) = \arg\max_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(\mathcal{D})}$$

We treat $\boldsymbol{\theta}$ as parameters (not random variables) that we need to set to have a high likelihood.

Examples: Getting expressions for the mean of a Gaussian. Or for learning the parameters of a neural network, in which case we often use:

$$\hat{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta}} \sum_{n=1}^{N} \log P_{\boldsymbol{\theta}}(y^{(n)}|\mathbf{x}^{(n)})$$

which is the same as minimising the negative log likelihood loss.

Bayesian modelling

$$\boxed{p(\boldsymbol{\theta}|\mathcal{D}) = \frac{p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta})}{p(\mathcal{D})} \propto p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta})}$$

We treat $\boldsymbol{\theta}$ as random variables and get a posterior distribution given the data.

The Bayesian approach says nothing about how we should use this distribution to make predictions.

But we can do the following to make a prediction for a new input $\mathbf{x}$:[3]

$$p(y|\mathbf{x}, \mathcal{D}) = \int_{\boldsymbol{\theta}} p(y, \boldsymbol{\theta}|\mathbf{x}, \mathcal{D}) \, \mathrm{d}\boldsymbol{\theta} = \int_{\boldsymbol{\theta}} p(y|\mathbf{x}, \boldsymbol{\theta}) p(\boldsymbol{\theta}|\mathcal{D}) \, \mathrm{d}\boldsymbol{\theta}$$

There are some arguments (Gal and Ghahramani 2015) that dropout can be seen as a way to represent uncertainty in neural networks (although this is apparently a bit controversial—and I don't know enough to comment on it).

Maximum a posteriori (MAP)

$$\boxed{\hat{\boldsymbol{\theta}}_{\mathrm{MAP}} = \arg\max_{\boldsymbol{\theta}} p(\boldsymbol{\theta}|\mathcal{D}) = \arg\max_{\boldsymbol{\theta}} p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta})}$$

A point summary of the posterior $p(\boldsymbol{\theta}|\mathcal{D})$.

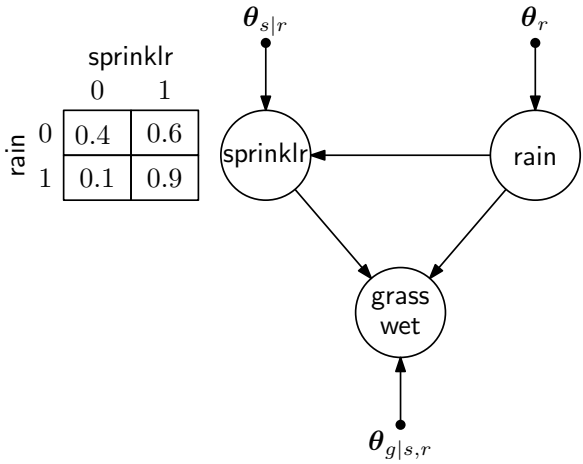
Many regularisation methods can be seen as a MAP estimate, e.g. L_1 and L_2 regularisation.

Maximum likelihood corresponds to MAP if we use a flat prior for the parameters $p(\boldsymbol{\theta})$.

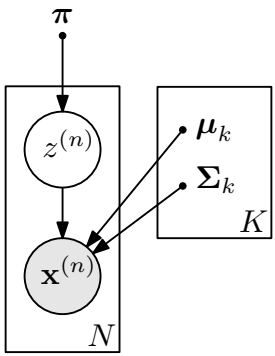
[3] See (Murray 2018a) and (Murray 2018b) respectively for regression and classification examples.

Learning 101: Expectation maximisation

If all variables are observed, maximum likelihood estimation is easy:



But if we have latent variables it is not so easy:



Chicken and egg problem:

- If we knew the parameters, then we could infer the latents.
- If we could observe the latents, then we could just do maximum likelihood.

Intuition: We guess initial parameters $\boldsymbol{\theta}^{(0)}$, infer the latents, do maximum likelihood to get new parameters $\boldsymbol{\theta}^{(1)}$, infer the latents, etc.

The expectation maximisation (EM) algorithm

- E-step: Determine

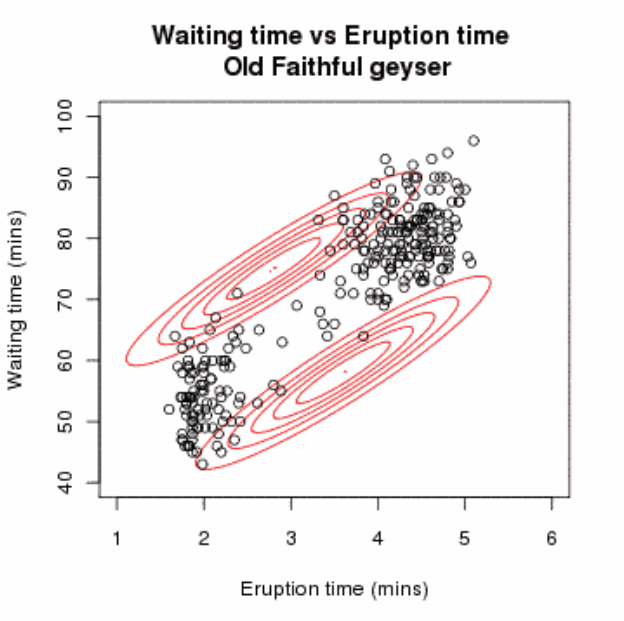
$$P_{\boldsymbol{\theta}^{(m)}}(\mathbf{z}^{(n)}|\mathbf{x}^{(n)})$$

for $n = 1, 2, \ldots, N$. This requires inference, which in some cases is easy (e.g. in a GMM) but in other cases requires some of the inference methods described below (e.g. in an HMM we need message passing).

- M-step:

$$\boldsymbol{\theta}^{(m+1)} = \arg\max_{\boldsymbol{\theta}} \sum_{n=1}^{N} \sum_{\mathbf{z}^{(n)}} P_{\boldsymbol{\theta}^{(m)}}(\mathbf{z}^{(n)}|\mathbf{x}^{(n)}) \log p_{\boldsymbol{\theta}}(\mathbf{x}^{(n)}, \mathbf{z}^{(n)})$$

GMM example:[4]



[4]Figure from Wikipedia.

More formally

For a full formal discussion, have a look at my EM note. I summarise it briefly.

We want to maximise the likelihood of our parameters:

$$\arg\max_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(\mathcal{D}) = \arg\max_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\mathcal{D})$$

Let's say we only have a single training item: $\mathcal{D} = \{\mathbf{x}^{(n)}\}$.[5]

$$\log p_{\boldsymbol{\theta}}(\mathbf{x}^{(n)}) = \log \sum_{\mathbf{z}^{(n)}} p_{\boldsymbol{\theta}}(\mathbf{x}^{(n)}, \mathbf{z}^{(n)})$$

The log-of-sum is irritating since it is difficult to differentiate.

So we get a lower bound that has sums-of-logs:

$$\log p_{\boldsymbol{\theta}}(\mathbf{x}^{(n)}) \geq -J(Q, \boldsymbol{\theta})$$

with

$$\begin{aligned} -J(Q, \boldsymbol{\theta}) &= \sum_{\mathbf{z}^{(n)}} Q(\mathbf{z}^{(n)}) \log p_{\boldsymbol{\theta}}(\mathbf{x}^{(n)}, \mathbf{z}^{(n)}) - \sum_{\mathbf{z}^{(n)}} Q(\mathbf{z}^{(n)}) \log Q(\mathbf{z}^{(n)}) \\ &= \mathbb{E}_Q \left[\log p_{\boldsymbol{\theta}}(\mathbf{x}^{(n)}|\mathbf{z}^{(n)})\right] + \mathbb{E}_Q \left[\log p_{\boldsymbol{\theta}}(\mathbf{z}^{(n)})\right] - \mathbb{E}_Q \left[\log Q(\mathbf{z}^{(n)})\right] \end{aligned}$$

$-J$ is called the *evidence lower bound* (ELBO) and Q is some helper distribution that we need to choose.

In classic EM (the one we are doing here) we set

$$Q(\mathbf{z}^{(n)}) = P_{\boldsymbol{\theta}^{(m)}}(\mathbf{z}^{(n)}|\mathbf{x}^{(n)})$$

(There is a really good reason for this: The bound becomes tight. Don't worry if you do not get this from just this description—you would need to go through the full EM note.)

The reason I am explaining all of this is that when we get to variational inference (which can be used for Bayesian models), you will see that we also optimise the ELBO but we set Q in a different way.

[5] I will do this a lot in this note. It is easy to extend to the full dataset.

EM as a black-box approach?

EM can be used for doing maximum likelihood estimation or MAP estimation when we have latent variables.

But EM cannot be applied to all models, e.g. it is difficult to use in this classic form for latent Dirichlet allocation (LDA).

So, although I really like EM, it cannot be used as the basis for a black-box framework in general.

Learning as inference

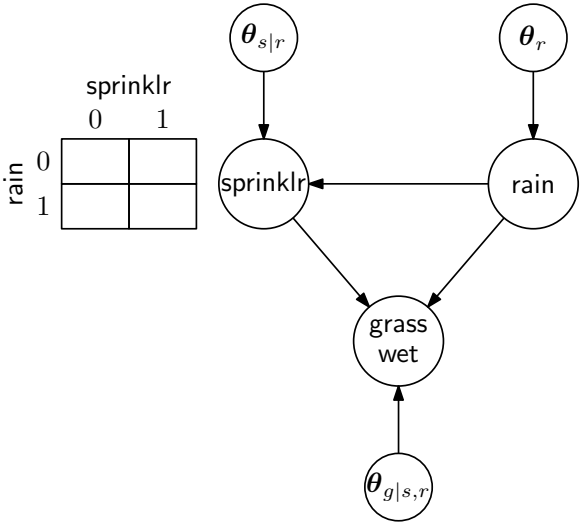
This is the title of Chap. 9 of (Barber 2020), and it is mentioned often in PGM courses. I never understood what this meant.

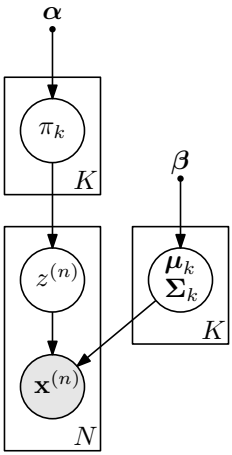
Inference: If we know the model and we have some observations, answering questions about some of the variables in the model.

Learning: Figuring out the parameters from observed data.

But what if we treat our parameters as random variables?

Being Bayesian: Our parameters as random variables





If we can answer questions about random variables in our model (inference) and our parameters are random variables, then we can answer questions about our parameters (learning) using inference methods. So we can learn by doing inference.

The grand goal of probabilistic modelling

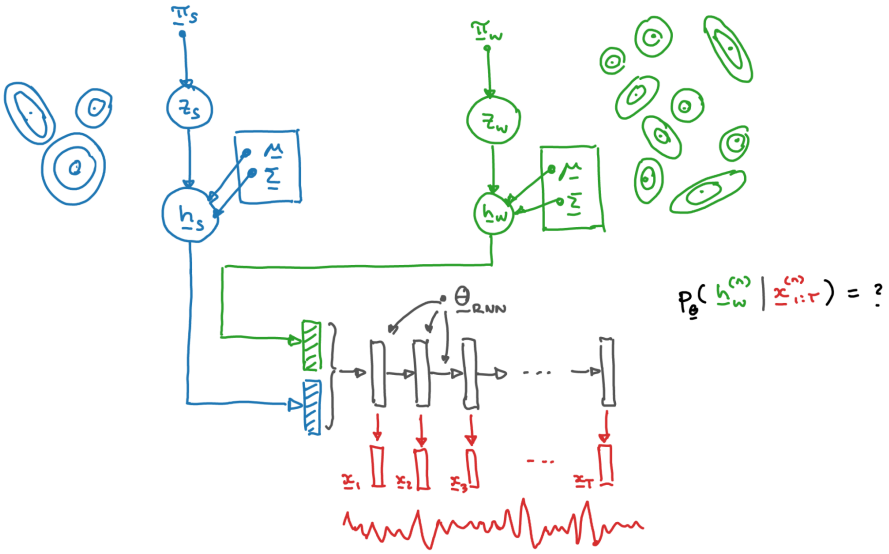
I want to draw, write or code my model and then have some black-box method do inference/learning for me.

E.g. I want to be able to code up either the Bayesian rain model or the Bayesian GMM above in the same framework and have that framework do inference/learning for both these models.

Example from Stan for linear regression:

```
data {
    int<lower=0> N;
    vector[N] x;
    vector[N] y;
}
parameters {
    real alpha;
    real beta;
    real<lower=0> sigma;
}
model {
    for (n in 1:N)
        y[n] ~ normal(alpha + beta * x[n], sigma);
}
```

Or I might want to do inference and learning in the following model, that I just made up:



(Disclaimer/spoiler: I don't think this is possible yet. If it is, please implement this model and come and show me.)

Seperating model from algorithm

Linked to the grand goal.

Shakir Mohamed often points out that it is useful to think of your model separately from the algorithm that you use to do inference/learning.

I.e. it is sometimes useful to just think about the model definition itself, before thinking about how you couple it with data via some inference/learning algorithm.

I have found this view helpful quite a few times when I was getting confused about some specific intricate details.

Three "classes" of inference/learning approaches

1. Belief propagation (message passing)
2. Sampling
3. Variational inference

The distinction between these are not always strict, e.g. variational message passing.

Belief propagation

Belief propagation is a type of message passing algorithm that is used to do inference in probabilistic graphical models (Wikipedia).

Different names for the same thing:

- Sum-product algorithm
- Shafer-Shenoy algorithm
- Belief propagation

Belief propagation can be applied on different types of representations. Factor graphs are useful since they can represent both Bayesian networks and Markov random fields. Cluster graphs is a generalisation of factor graphs.

At Stellenbosch University, if someone says they are working with PGMs, this probably means that they are doing something with belief propagation.

Factor graphs

The summary of factor graphs on Wikipedia is pretty good.

The joint density for a factor graph:

$$p(x_1, x_2, \ldots, x_D) = \frac{1}{Z} \prod_i f_i(\mathcal{X}_i)$$

where $\mathcal{X}_i$ is the set of variables that factor f_i operates on and Z is a normalisation constant.

Graphically:

- An undirected graph.
- Variable nodes are round.
- Factor nodes are rectangular.

All the variables associated with a factor are neighbours to it.

Any Bayesian network or Markov random field can be represented as a factor graph.

Example: Converting a Bayesian networks to a factor graph

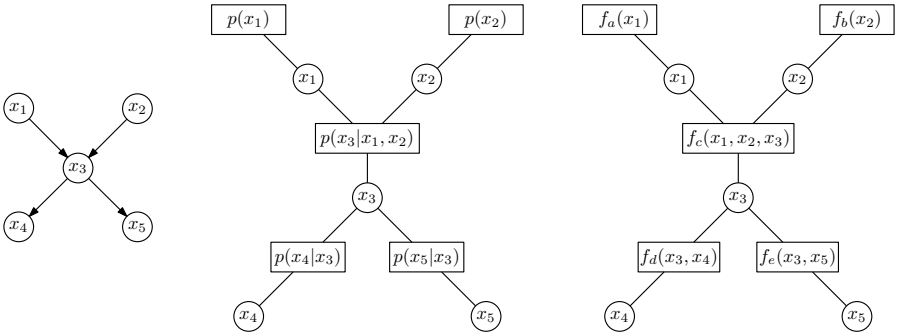
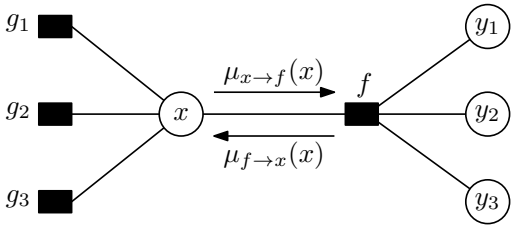


Figure adapted from (Murphy 2012, Fig. 22.3)

Belief propagation on a factor graph

Messages of real valued functions are passed between nodes (Barber 2022, Proc. 5.1):



- Initialisation:
  - Messages from leaf factor nodes: Set to factor.
  - Message from leaf variable nodes: Set to unity.
- Variable-to-factor messages:

$$\mu_{x\to f}(x) = \prod_{g\in\{\mathrm{ne}(x)\backslash f\}} \mu_{g\to x}(x)$$

- Factor-to-variable messages:

$$\mu_{f\to x}(x) = \sum_{\mathcal{Y}} f(x,\mathcal{Y}) \prod_{y\in\{\mathrm{ne}(f)\backslash x\}} \mu_{y\to f}(y)$$

  where $\mathcal{Y} = \{\mathrm{ne}(f)\backslash x\}$, so $\sum_{\mathcal{Y}}$ marginalises out all the variables of factor f apart from x (e.g. marginalising out y_1, y_2, y_3).

  Messages are only sent after all incoming messages are received.

- Termination:
  - Marginal for singly connected graphs (i.e. trees):

$$p(x) \propto \prod_{f\in\mathrm{ne}(x)} \mu_{f\to x}(x)$$

  - Belief for loopy graphs:

$$\mathrm{bel}(x) \propto \prod_{f\in\mathrm{ne}(x)} \mu_{f\to x}(x)$$

Belief propagation is exact in trees

- We get the exact true probabilities.
- Only need to pass messages once.
- (Chains are also trees.)

You have actually already seen belief propagation: It is a generalisation of the forward-backward algorithm used in HMMs.

What if we have loops?

1. Convert graph to a tree: Junction tree algorithm. In the extreme case you end up with a single factor with all the variables, which means you don't get any saving from belief propagation compared to just explicitly evaluating the full joint distribution.

2. Use your loopy graph, pretend you don't notice, and just do belief propagation: Loopy belief propagation.

    - Multiple rounds of message passing (with the order having an effect).
    - The result is approximate.

Dealing with evidence

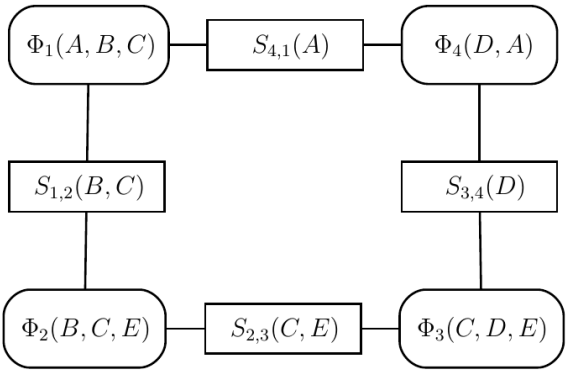
Two options (Barber 2020, Sec. 5.1.3):

1. Set evidence variables $\mathcal{X}_e$ to observed states, define new factors, do belief propagation.

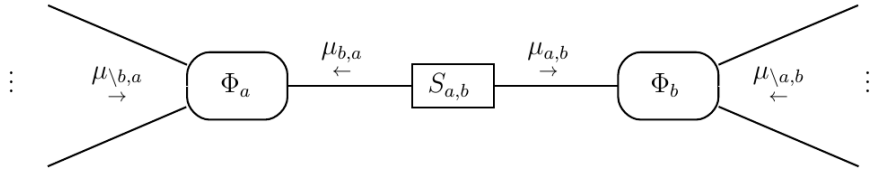
2. Multiply factor with an indicator function that sets anything to zero if an evidence variable x_e is not in its observed state.

Cluster graphs

Cluster graphs is a generalisation of factor graphs.[6]



Can also formulate belief propagation in terms of cluster graphs:



There are also variations and extensions of loopy belief propagation on cluster graphs: Loopy belief update (Lauritzen-Spiegelhalter). See (Du Preez 2022a, Sec. 19.1).

Stellenbosch University people really like cluster graphs.

[6] Figures by Werner van der Merwe.

Thoughts on belief propagation

Works well when we have a small number of variables and have a good idea of how they interact (independencies).

In these cases (known unknowns), PGM representations can be way more efficient than storing the full joint.

It is also useful to know about it as a generalisation of forward-backward.

But I suspect it is difficult to scale if we have millions of parameters where we are unsure of how they interact (unknown unknowns).

We have really only looked above at the case for discrete variables: What do you do if they are continuous? Gaussian belief propagation.

Sampling

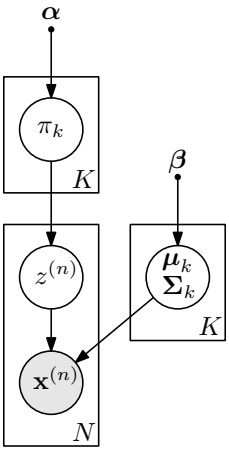
Why sampling?

We can approximate expected values (Resnik and Hardisty 2010):

$$\mathbb{E}\left[f(\boldsymbol{\theta})\right] \approx \frac{1}{L}\sum_{l=1}^{L} f(\boldsymbol{\theta}^{(l)})$$

where $\boldsymbol{\theta}^{(l)} \sim p(\boldsymbol{\theta})$ are samples from $p(\boldsymbol{\theta})$.

Example: Bayesian GMM



In this model we have the following random variables (some observed):

- Observed data: $\mathcal{X} = \{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(N)}\}$
- Latent component assignments: $\mathbf{z} = \{z^{(1)}, z^{(2)}, \ldots z^{(N)}\}$
- Mixture weights: $\boldsymbol{\pi} = [\pi_1, \pi_2, \ldots, \pi_K]$
- Component means: $\{\boldsymbol{\mu}_k\}_{k=1}^K = \{\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \ldots, \boldsymbol{\mu}_K\}$
- Component covariance matrices: $\{\boldsymbol{\Sigma}_k\}_{k=1}^K = \{\boldsymbol{\Sigma}_1, \boldsymbol{\Sigma}_2, \ldots, \boldsymbol{\Sigma}_K\}$

We can bunch all these together:

$$\boldsymbol{\theta} = \left\{\mathcal{X}, \mathbf{z}, \boldsymbol{\pi}, \{\boldsymbol{\mu}_k\}_{k=1}^K, \{\boldsymbol{\Sigma}_k\}_{k=1}^K\right\}$$

Now say we want to know the assignment of observed item $\mathbf{x}^{(n)}$. Then we can use the above sampling equations with $f(\boldsymbol{\theta}) = z^{(n)}$ to approximate $\mathbb{E}\left[z^{(n)}\right]$, i.e. the expected component to which item $\mathbf{x}^{(n)}$ is assigned.

This seems simple enough, but we need to actually be able to sample from $p(\boldsymbol{\theta})$ in order to do this.

We look at two methods. Both are instances of Markov chain Monte Carlo (MCMC).

Gibbs sampling

Core idea: Cycle through variables and sample in turn from $p(\theta_i|\boldsymbol{\theta}_{\setminus i})$.

E.g, if we have three variables θ_1, θ_2, and θ_3 then we would sample:

$$\begin{aligned}
\theta_1^{(s+1)} &\sim p(\theta_1|\theta_2^{(s)}, \theta_3^{(s)}) \\
\theta_2^{(s+1)} &\sim p(\theta_2|\theta_1^{(s+1)}, \theta_3^{(s)}) \\
\theta_3^{(s+1)} &\sim p(\theta_3|\theta_1^{(s+1)}, \theta_2^{(s+1)})
\end{aligned}$$

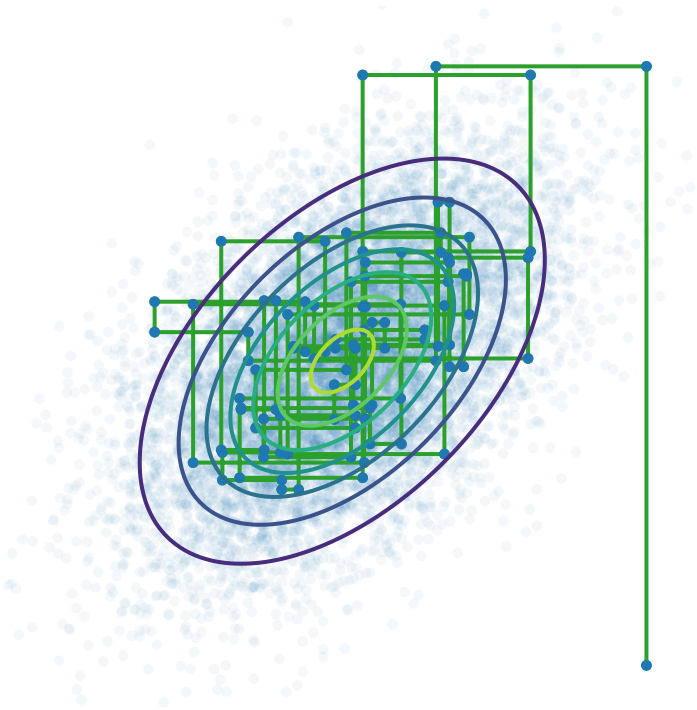
So in general we cycle through the variables and sample in turn from

$$\theta_i^{(s+1)} \sim p(\theta_i|\theta_1^{(s+1)}, \ldots, \theta_{i-1}^{(s+1)}, \theta_{i+1}^{(s)}, \ldots, \theta_D^{(s)})$$

Note that we still need to have expressions for and be able to sample from all of these $p(\theta_i|\boldsymbol{\theta}_{\setminus i})$ distributions. To get these, it often helps to use conjugate priors.

Gibbs example: 2D Gaussian

The mean and covariance are known parameters.



Gibbs example: Bayesian GMM

We would need to be able to sample from

$$P(z^{(n)} = k|\mathbf{z}_{\setminus n}, \mathcal{X}, \boldsymbol{\pi}, \{\boldsymbol{\mu}_k\}_{k=1}^K, \{\boldsymbol{\Sigma}_k\}_{k=1}^K)$$

Collapsed and blocked Gibbs sampling

Collapsed Gibbs sampling

In some cases we are able to analytically integrate out some of the unknown variables and then just need to sample the rest.

E.g. in the Bayesian GMM, if we choose the priors $p_{\boldsymbol{\alpha}}(\boldsymbol{\pi})$ and $p_{\beta}(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$ carefully (specifically that they are conjugate priors to some of the other distributions), then we just have to sample the component assignments

$$P(z^{(n)}|\mathbf{z}_{\setminus n}, \mathcal{X})$$

You can get the details in my note on Bayesian GMMs. Also see the corresponding demo that makes use of collapsed sampling.

Blocked Gibbs sampling

In some cases we can sample groups of variables together:

$$\begin{aligned}
\theta_1^{(s+1)}, \theta_2^{(s+1)} &\sim p(\theta_1, \theta_2 | \theta_3^{(s)}, \theta_4^{(s)}, \theta_5^{(s)}, \theta_6^{(s)}) \\
\theta_3^{(s+1)}, \theta_4^{(s+1)} &\sim p(\theta_3, \theta_4 | \theta_1^{(s+1)}, \theta_2^{(s+1)}, \theta_5^{(s)}, \theta_6^{(s)}) \\
\theta_5^{(s+1)}, \theta_6^{(s+1)} &\sim p(\theta_5, \theta_6 | \theta_1^{(s+1)}, \theta_2^{(s+1)}, \theta_3^{(s+1)}, \theta_4^{(s+1)})
\end{aligned}$$

Metropolis-Hastings

Gibbs sampling is actually a special case of the Metropolis-Hastings algorithm (and both are instances of MCMC).

The Metropolis-Hastings algorithm:

- Sample a proposed move from a proposal distribution:

$$\boldsymbol{\theta}' \sim q(\boldsymbol{\theta}'; \boldsymbol{\theta}^{(s)})$$

- Do we accept the proposal?

$$P(\texttt{accept}) = \min\left(1, \frac{p(\boldsymbol{\theta}')\, q(\boldsymbol{\theta}^{(s)}; \boldsymbol{\theta}')}{p(\boldsymbol{\theta}^{(s)})\, q(\boldsymbol{\theta}'; \boldsymbol{\theta}^{(s)})}\right)$$

- if `accept`:

$$\boldsymbol{\theta}^{(s+1)} = \boldsymbol{\theta}'$$

  else:

$$\boldsymbol{\theta}^{(s+1)} = \boldsymbol{\theta}^{(s)}$$

Reminder: We are trying to sample from the true unknown distribution $p(\boldsymbol{\theta})$. Note that above we do not need to know this distribution, but we need to be able to evaluate it at a specific point, i.e. we need to be able to get the height of the PDF at a given sample value.

To understand the acceptance probability better, let's look at the symmetric case where $q(\boldsymbol{\theta}_a; \boldsymbol{\theta}_b) = q(\boldsymbol{\theta}_b; \boldsymbol{\theta}_a)$:

$$P(\texttt{accept}) = \min\left(1, \frac{p(\boldsymbol{\theta}')}{p(\boldsymbol{\theta}^{(s)})}\right)$$

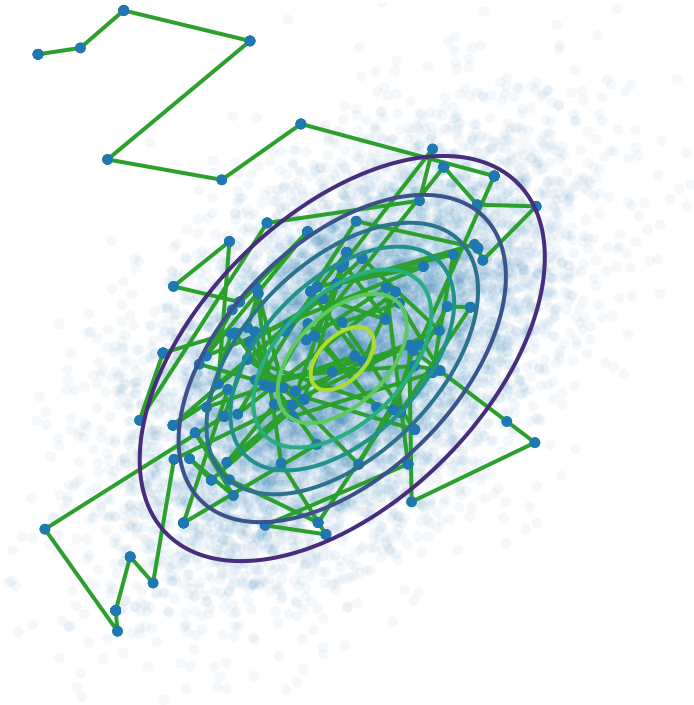
So if the proposal is better than the current sample according to $p(\boldsymbol{\theta})$ (the true unknown distribution that we are trying to sample from), then we definitely move. Otherwise we move proportional to how the probability of the proposal compares to that of the current sample.

A popular proposal distribution:

$$q(\boldsymbol{\theta}'; \boldsymbol{\theta}^{(s)}) = \mathcal{N}(\boldsymbol{\theta}'; \boldsymbol{\theta}^{(s)}, \boldsymbol{\Sigma})$$

where $\boldsymbol{\Sigma}$ is a hyperparameter that we need to set.

Metropolis-Hastings example: 2D Gaussian



Sampling in practice

We want a black-box method where we can draw, write, or code a model and then have a general purpose package do inference/learning for us.

A number of probabilistic programming languages have been under development for a while now and are continuing to be developed. Several of them use MCMC-based sampling:

- BUGS
- JAGS
- Stan

My (very limited) experience of these languages matches that of Murphy's (2012, Sec. 24.2.6):

> Although this approach is appealing, unfortunately it can be much slower than using hand-written code, especially for complex models.

Variational inference

The terminology (VI, VB, VBEM, VMP) that I use here roughly follows that of Murphy (2012, Chap. 21).

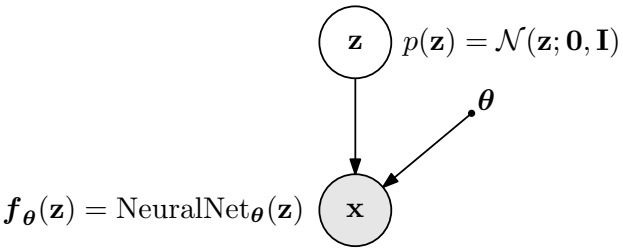
These methods are called *variational* since we optimise over functions: Instead of just having parameters that we change to minimise a loss function, we also have functions that we can themself change to minimise the loss function. (In practice, we normally still end up just optimising the parameters of these variational functions.)

Variational inference (the Pyro setup)

We start with a setting that isn't fully Bayesian (yet), but very common. This setting is followed in Pyro. We have a model with:

- Latent variables $\mathbf{z}$ for which we have a prior distribution.
- Parameters $\boldsymbol{\theta}$ for which we want to get a point estimate, probably using maximum likelihood. (If we were Bayesian, we would have defined a prior over these parameters and tried to get a distribution over them instead of a point estimate.)

One example of exactly this setting is in a variational autoencoder:



We optimise a model by pushing up a lower bound called the *evidence lower bound* (ELBO). We have seen this before when I introduced EM.

For now, let's say we have a dataset with a single training point $\mathbf{x}^{(n)}$. Then we have (following the procedure in my EM note):

$$\log p_{\boldsymbol{\theta}}(\mathbf{x}^{(n)}) \geq \mathbb{E}_q \left[\log p_{\boldsymbol{\theta}}(\mathbf{x}^{(n)}|\mathbf{z}^{(n)})\right] + \mathbb{E}_q \left[\log p_{\boldsymbol{\theta}}(\mathbf{z}^{(n)})\right] - \mathbb{E}_q \left[\log q(\mathbf{z}^{(n)})\right] = -J(q, \boldsymbol{\theta})$$

Here $q(\mathbf{z})$ is a helper distribution and $-J(q, \boldsymbol{\theta})$ is the ELBO. By maximising the ELBO, the hope is that we will get a $\boldsymbol{\theta}$ with a high log likelihood $\log p_{\boldsymbol{\theta}}(\mathbf{x}^{(n)})$.

The ELBO can also be written as:

$$-J(q, \boldsymbol{\theta}) = -D_{\mathrm{KL}}\left(q(\mathbf{z}^{(n)}) \,\|\, p_{\boldsymbol{\theta}}(\mathbf{z}^{(n)}|\mathbf{x}^{(n)})\right) + \log p_{\boldsymbol{\theta}}(\mathbf{x}^{(n)})$$

This helps us to see that, when we are maximising the ELBO, then we are really trying to find a $q(\mathbf{z})$ that approximates the posterior $p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})$.

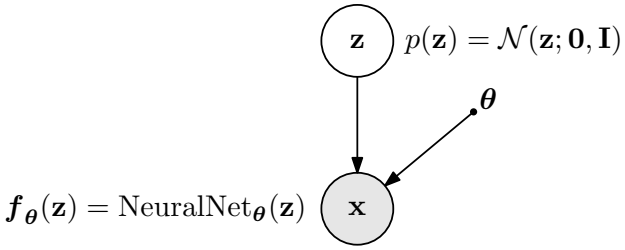
So far we have been following exactly the same route as in classical EM:

- In the E-step in classic EM, we would set $q(\mathbf{z}^{(n)}) = p_{\boldsymbol{\theta}^{(m)}}(\mathbf{z}^{(n)}|\mathbf{x}^{(n)})$ using the current estimate of $\boldsymbol{\theta}^{(m)}$.
- In general variational inference, we instead optimise the ELBO by specifying a useful form for $q_{\boldsymbol{\phi}}(\mathbf{z})$ (e.g. Gaussian) and then find the optimal parameters $\boldsymbol{\phi}$. So we minimise $-J(\boldsymbol{\phi}, \boldsymbol{\theta})$.
- (Again, this is exactly what happens in a variational autoencoder.)

We call $\boldsymbol{\phi}$ the variational parameters and $q_{\boldsymbol{\phi}}(\mathbf{z})$ the variational distribution. In Pyro, they call $q_{\boldsymbol{\phi}}(\mathbf{z})$ the *guide*, since it guides us towards the intractable posterior.

Note that in classical EM we are guaranteed to improve $\log p_{\boldsymbol{\theta}}(\mathbf{x}^{(n)})$ at every step because the bound is tight. This is not the case in general variational inference.

Applied to the variational autoencoder



We often choose:

$$p(\mathbf{z}) = \mathcal{N}(\mathbf{z}; \mathbf{0}, \mathbf{I})$$
$$p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z}) = \mathcal{N}(\mathbf{x}; \boldsymbol{f}_{\boldsymbol{\theta}}(\mathbf{z}), \sigma^2\mathbf{I})$$

We want to maximise the log likelihood by changing $\boldsymbol{\theta}$:

$$\log p_{\boldsymbol{\theta}}(\mathbf{x}^{(n)}) = \log \int_{\mathbf{z}} p_{\boldsymbol{\theta}}(\mathbf{x}^{(n)}|\mathbf{z}) p(\mathbf{z}) \, \mathrm{d}\mathbf{z}$$

Even though we have chosen specific forms for $p(\mathbf{z})$ and $p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})$ (so it is easy to go forward), the marginalisation really messes up optimisation: you end up having to take derivates of a log-of-a-sum/integral, which is much harder than taking the derivative of a sum-of-logs.

So instead, we maximise the ELBO.

This introduces a guiding distribution $q_{\boldsymbol{\phi}}(\mathbf{z})$. What this does is really serve as an approximation for the tricky $p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})$. So we can write (informally):

$$q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) \approx p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})$$

We have not solved everything by doing all of the above: The ELBO still has some tricky terms (one in particular)!

Now back to looking at more general cases, and we will solve the tricky term along the way.

Variational Bayes

A setting where we can be fully Bayesian and follow exactly the same steps as in the Pyro setup above, is one where we have a model without any latent variables and parameters $\boldsymbol{\theta}$ which we now treat as random variables.

In this case we get the ELBO by marginalising out the parameters:

$$\begin{aligned}\log p(\mathbf{x}^{(n)}) &= \int_{\boldsymbol{\theta}} p(\mathbf{x}^{(n)}, \boldsymbol{\theta}) \, \mathrm{d}\boldsymbol{\theta} \\ &\geq \mathbb{E}_q \left[\log p(\mathbf{x}^{(n)} | \boldsymbol{\theta}) \right] + \mathbb{E}_q \left[\log p(\boldsymbol{\theta}) \right] - \mathbb{E}_q \left[\log q_{\boldsymbol{\phi}}(\boldsymbol{\theta}) \right]\end{aligned}$$

A very illustrative example in this setting is given in (Murray 2018c) and (Murray 2018d) for Bayesian logistic regression.

Stochastic variational inference (SVI)

The tricky term in the ELBO: $\mathbb{E}_q \left[\log p_{\boldsymbol{\theta}}(\mathbf{x}^{(n)}|\mathbf{z}^{(n)})\right]$

In SVI, we approximate this term by sampling (Resnik and Hardisty 2010):

$$\mathbb{E}_q \left[\log p_{\boldsymbol{\theta}}(\mathbf{x}^{(n)}|\mathbf{z}^{(n)})\right] \approx \frac{1}{L}\sum_{l=1}^{L} \log p_{\boldsymbol{\theta}}(\mathbf{x}^{(n)}|\mathbf{z}^{(l)})$$

where $\mathbf{z}^{(l)} \sim q_{\boldsymbol{\phi}}(\mathbf{z})$ are samples from $q_{\boldsymbol{\phi}}(\mathbf{z})$. We often use a single sample $L = 1$.

For the variational Bayes setting above without latent variables, we would similarly have

$$\mathbb{E}_q \left[\log p(\mathbf{x}^{(n)}|\boldsymbol{\theta})\right] \approx \frac{1}{L}\sum_{l=1}^{L} \log p(\mathbf{x}^{(n)}|\boldsymbol{\theta}^{(l)})$$

where $\boldsymbol{\theta}^{(l)} \sim q_{\boldsymbol{\phi}}(\boldsymbol{\theta})$ are samples from $q_{\boldsymbol{\phi}}(\boldsymbol{\theta})$.

Doing this allows us to use the *reparametrisation trick*, which allows for gradient calculation if we are using an automatic differentiation framework. This is one of the key things that are enabling black-box inference in toolkits like Pyro. And (again!) this is also what is done in a variational autoencoder. To understand this, let's look at an example.

The reparametrisation trick

The guide: We look at a (somewhat) concrete example, where we use a diagonal Gaussian as our guide $q_{\boldsymbol{\phi}}(\mathbf{z})$. The guide $q_{\boldsymbol{\phi}}(\mathbf{z})$ also depends on $\mathbf{x}$ since it is an approximation for $p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})$. So I will write $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$. We obtain the parameters of the diagonal Gaussian from $\mathbf{x}$, by passing it through a small neural network with parameters $\boldsymbol{\phi}$. With all of this, we can write our guide as

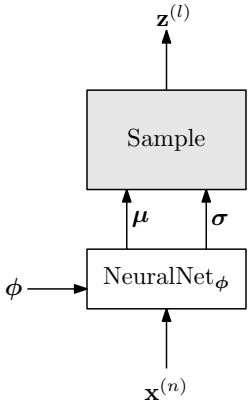
$$\begin{aligned}\boldsymbol{\mu}, \log \boldsymbol{\sigma} &= \mathrm{NeuralNet}_{\boldsymbol{\phi}}(\mathbf{x}) \\ q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) &= \mathcal{N}(\mathbf{z}|\boldsymbol{\mu}, \mathrm{diag}(\boldsymbol{\sigma}^2))\end{aligned}$$

Optimisation: We want to optimise $J(\boldsymbol{\phi}, \boldsymbol{\theta})$ in terms of both $\boldsymbol{\phi}$ and $\boldsymbol{\theta}$. Above we showed that we can deal with the tricky term in the ELBO using sampling. E.g. we could estimate the gradients for $\boldsymbol{\phi}$:

$$\frac{\partial}{\partial \boldsymbol{\phi}} \mathbb{E}_q \left[\log p_{\boldsymbol{\theta}}(\mathbf{x}^{(n)}|\mathbf{z}^{(n)}) \right] \approx \frac{1}{L} \sum_{l=1}^{L} \frac{\partial}{\partial \boldsymbol{\phi}} \log p_{\boldsymbol{\theta}}(\mathbf{x}^{(n)}|\mathbf{z}^{(l)})$$

where $\mathbf{z}^{(l)} \sim q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}^{(n)})$.

This all seems fine, but the problem is that there is sampling involved in getting $\mathbf{z}^{(l)}$. How do we get the gradient through a sampling block where some randomness happens?!



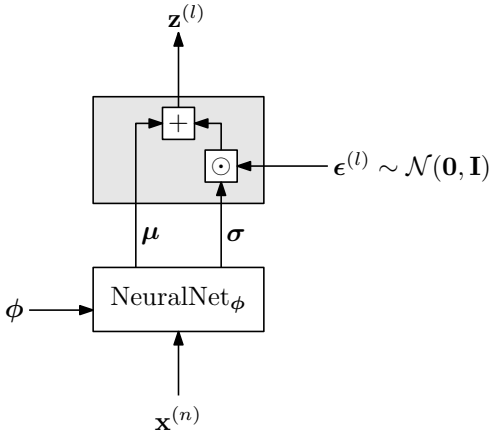
The trick: We take the randomness out of the block.

Formally, we express $\mathbf{z}^{(l)} \sim q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}^{(n)})$ as a differentiable transformation of a sample $\boldsymbol{\epsilon}^{(l)}$ from another distribution that is independent of both $\boldsymbol{\phi}$ and $\mathbf{x}$ (Mouton 2023):

$$\mathbf{z}^{(l)} = g_{\boldsymbol{\phi}}(\boldsymbol{\epsilon}^{(l)}, \mathbf{x})$$

In our example:

$$\mathbf{z}^{(l)} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}^{(l)} \text{ where } \boldsymbol{\epsilon}^{(l)} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$$



Now we can get the gradients through the sampling block!

Variational Bayesian expectation maximisation (VBEM)

An alternative to SVI.

Used in models where we have both latent variables and parameters which are now treated as random variables and then marginalised out (i.e. we are really Bayesian now).

In this case the ELBO is:

$$\begin{aligned}\log p(\mathbf{x}^{(n)}) &= \log \int_{\mathbf{z}} \int_{\boldsymbol{\theta}} p(\mathbf{x}^{(n)}, \mathbf{z}, \boldsymbol{\theta}) \, \mathrm{d}\boldsymbol{\theta} \, \mathrm{d}\mathbf{z} \\ &\geq \int_{\mathbf{z}} \int_{\boldsymbol{\theta}} q(\mathbf{z}, \boldsymbol{\theta}) \log \frac{p(\mathbf{x}^{(n)}, \mathbf{z}, \boldsymbol{\theta})}{q(\mathbf{z}, \boldsymbol{\theta})} \, \mathrm{d}\boldsymbol{\theta} \, \mathrm{d}\mathbf{z}\end{aligned}$$

In VBEM we use the *mean-field assumption*:

$$q(\mathbf{z}, \boldsymbol{\theta}) = q(\mathbf{z}) q(\boldsymbol{\theta})$$

The ELBO then becomes

$$\begin{aligned}-J(q) &= \int_{\mathbf{z}} \int_{\boldsymbol{\theta}} q(\mathbf{z}, \boldsymbol{\theta}) \log \frac{p(\mathbf{x}^{(n)}, \mathbf{z}, \boldsymbol{\theta})}{q(\mathbf{z}, \boldsymbol{\theta})} \, \mathrm{d}\boldsymbol{\theta} \, \mathrm{d}\mathbf{z} \\ &= \int_{\mathbf{z}} \int_{\boldsymbol{\theta}} q(\mathbf{z}) q(\boldsymbol{\theta}) \log \frac{p(\mathbf{x}^{(n)}, \mathbf{z}, \boldsymbol{\theta})}{q(\mathbf{z}) q(\boldsymbol{\theta})} \, \mathrm{d}\boldsymbol{\theta} \, \mathrm{d}\mathbf{z} \\ &= \mathbb{E}_{q(\mathbf{z})q(\boldsymbol{\theta})} \left[\log p(\mathbf{x}^{(n)}, \mathbf{z}, \boldsymbol{\theta}) \right] - \mathbb{E}_{q(\mathbf{z})} \left[\log q(\mathbf{z}) \right] - \mathbb{E}_{q(\boldsymbol{\theta})} \left[\log q(\boldsymbol{\theta}) \right]\end{aligned}$$

In VBEM we now alternate between two steps:

- E-step: Keep $q(\boldsymbol{\theta})$ fixed and find $q(\mathbf{z})$.
- M-step: Keep $q(\mathbf{z})$ fixed and find $q(\boldsymbol{\theta})$.

Barber (2020, Alg. 11.4) shows that with the ELBO as above we get the following EM algorithm:

- E-step:

$$\begin{aligned} q^{(\mathsf{new})}(\mathbf{z}) &= \arg\max_{q(\mathbf{z})} \left\{ \mathbb{E}_{q(\mathbf{z}) q^{(\mathsf{old})}(\boldsymbol{\theta})} \left[\log p(\mathbf{x}^{(n)}, \mathbf{z}, \boldsymbol{\theta}) \right] - \mathbb{E}_{q(\mathbf{z})} \left[\log q(\mathbf{z}) \right] \right\} \\ &\propto \exp \left\{ \log \mathbb{E}_{q^{(\mathsf{old})}(\boldsymbol{\theta})} \left[\log p(\mathbf{x}^{(n)}, \mathbf{z} | \boldsymbol{\theta}) \right] \right\} \end{aligned}$$

- M-step:

$$\begin{aligned} q^{(\mathsf{new})}(\boldsymbol{\theta}) &= \arg\max_{q(\boldsymbol{\theta})} \left\{ \mathbb{E}_{q^{(\mathsf{new})}(\mathbf{z}) q(\boldsymbol{\theta})} \left[\log p(\mathbf{x}^{(n)}, \mathbf{z}, \boldsymbol{\theta}) \right] - \mathbb{E}_{q(\boldsymbol{\theta})} \left[\log q(\boldsymbol{\theta}) \right] \right\} \\ &\propto p(\boldsymbol{\theta}) \exp \left\{ \log \mathbb{E}_{q^{(\mathsf{new})}(\mathbf{z})} \left[\log p(\mathbf{x}^{(n)}, \mathbf{z} | \boldsymbol{\theta}) \right] \right\} \end{aligned}$$

Variational message passing (VMP)

Mentioned briefly by Murphy (2012, Sec. 21.7).

This is an alternative to VBEM.

Can be used with directed graphical models where all the conditional probability distributions are in the exponential family.

You sweep over the graph, updating nodes one at a time, very similar to Gibbs sampling.

Thoughts on variational inference

The math can get quite hairy (compared to sampling).

SVI is where neural networks are starting to meet PGMs. But has this made a massive difference?

Conclusion

A lot of this you have probably seen in some other context before.

It is cool that PGMs generalise and draw things together.

And, after going through all this, it definitely impacts the way I look at some of my existing work, e.g. quickly drawing out a Bayesian network to see what other insights I can get from it.

But ...

At the same time it feels like in practice you still end up having to derive and implement most things specifically for a particular model.

Completely different issue: Is it always "better" to be Bayesian? In a lot of my reading this comes through implicitly, and this is a big question to ask.

(I like MAP and will try and use it more/think of my models more in terms of MAP estimation.)

References

D. Barber, *Bayesian Reasoning and Machine Learning*, 2020.

J. A. du Preez, *EMDW Development Guide*, 2022a.

J. A. du Preez, "PMR817: Probabilistic modelling and reasoning," *Stellenbosch University*, 2022b.

Y. Gal, Z. Ghahramani, "Dropout as a Bayesian approximation: Representing model uncertainty in deep learning," https://arxiv.org/abs/1506.02142, 2015.

H. Kamper, "Gibbs sampling for fitting finite and infinite Gaussian mixture models," *University of Edinburgh*, 2015.

H. Kamper, "Expectation maximisation," *Stellenbosch University*, 2022.

D. Koller and N. Friedman, *Probabilistic Graphical Models: Principles and Techniques*, 2009.

S. Mohamed, "Variational inference for machine learning," *Imperial College*, 2015.

J. Mouton, "Integrating Bayesian network structure into normalizing flows and variational autoencoders," Master's thesis, *Stellenbosch University*, 2023.

K. P. Murphy, *Machine Learning: A Probabilistic Perspective*, 1st ed., 2012.

I. Murray, "Monte Carlo inference methods," *NeurIPS Tutorial*, 2015.

I. Murray, "Bayesian regression," *University of Edinburgh*, 2018a.

I. Murray, "Bayesian logistic regression and Laplace approximations," *University of Edinburgh*, 2018b.

I. Murray, "Variational objectives and KL divergence," *University of Edinburgh*, 2018c.

I. Murray, "More details on variational methods," *University of Edinburgh*, 2018d.

P. Resnik and E. Hardisty, "Gibbs sampling for the uninitiated," *University of Maryland*, 2010.