Variational Inference and Optimization II

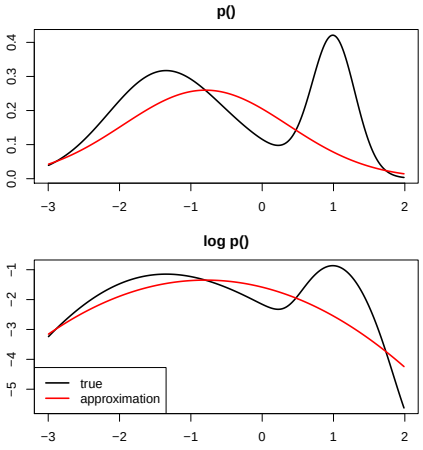
Arto Klami
University of Helsinki, Finland

ProbAI, June 16, 2021

Distributional approximation: Variational approximation

Variational inference directly solves for a suitable distribution that is close to the posterior:

- Postulate a parametric form $q(\boldsymbol{\theta}|\boldsymbol{\lambda})$
- Optimize for $\boldsymbol{\lambda}$ such that $q(\boldsymbol{\theta}|\boldsymbol{\lambda}) \approx p(\boldsymbol{\theta}|\mathcal{D})$
- ...by minimizing a suitable distance measure $d(q(\boldsymbol{\theta}|\boldsymbol{\lambda}), p(\boldsymbol{\lambda}|\mathcal{D})$ and making sure $\int q(\boldsymbol{\theta}|\boldsymbol{\lambda}) d\boldsymbol{\theta} = 1$



Important:

- $\boldsymbol{\theta}$ always refers here to parameters of the model, which are unknown – we want to learn the distribution over them
- $\boldsymbol{\lambda}$ are parameters of our approximation – we optimize over these

Classical VI recap

Standard VI uses Kullback-Leibler divergence between $q(\boldsymbol{\theta}|\boldsymbol{\lambda})$ and $p(\boldsymbol{\theta}|\mathcal{D})$ as the loss function

Equivalent formulation: Maximize a lower bound $\mathcal{L}_{ELBO}(\boldsymbol{\lambda})$ for $\log p(\mathcal{D})$:

$$\begin{aligned}\mathcal{L}_{ELBO}(\boldsymbol{\lambda}) &:= \mathbb{E}_{q(\boldsymbol{\theta}|\boldsymbol{\lambda})}\left[\log p(\mathcal{D}|\boldsymbol{\theta})\right] - D_{KL}(q(\boldsymbol{\theta}|\boldsymbol{\lambda}))|p(\boldsymbol{\theta})) \\ &= \mathbb{E}_{q(\boldsymbol{\theta}|\boldsymbol{\lambda})}\left[\log p(\mathcal{D}, \boldsymbol{\theta})\right] + \mathcal{H}(q(\boldsymbol{\theta}|\boldsymbol{\lambda})) \leq \log p(\mathcal{D})\end{aligned}$$

Classical VI recap

Mean-field approximation $q(\boldsymbol{\theta}) = \prod_{d=1}^{D} q_d(\boldsymbol{\theta}_d)$ leads to very elegant coordinate ascent algorithm with

$$q_d(\boldsymbol{\theta}_d) \propto e^{\mathbb{E}_{q_{-d}(\boldsymbol{\theta})}[\log p(\boldsymbol{x}, \boldsymbol{\theta})]}$$

or equivalently

$$q_d(\boldsymbol{\theta}_d) \propto e^{\mathbb{E}_{q_{-d}(\boldsymbol{\theta})}[\log p(\boldsymbol{\theta}_d | \boldsymbol{x}, \boldsymbol{\theta}_{-d})]}$$

If these conditionals are in exponential family (with conjugate priors), we get simple update rules that are deterministic functions of current parameters of other approximating terms

Pseudo-code for our example model from yesterday:

1. Initialize $q(\mu|m,t)$ and $q(\tau|\alpha,\beta)$ by setting some valid parameters for $\lambda = \{m, t, \alpha, \beta\}$
2. Update α and β with equations that only depend on $\mathcal{D}$, m and t
3. Update m and t with equations that only depend on $\mathcal{D}$, α and β
4. Repeat 2 and 3 until the objective no longer improves

Classical VI recap

Despite the nice analytic updates, the classical approach has serious limitations

- Slow and error-prone derivations
- Limited to mean-field approximation
- Limited to conditionally conjugate models

While some deviations are possible (see e.g. Jordan et al. [1999], Seeger and Bouchard [2012], Polson et al. [2013], Klami [2014] for different ways of handling logistic transformations), they are highly model-specific and still more complex

Model-free optimization

Rather than repeating the derivations for all new models and constraining the choice of models and approximations, we would want inference algorithms that are (more or less) independent of the model and approximation

This parallels the transition that has happened in deep learning, where tedious manual backpropagation calculations and implementing conjugate gradient descent from scratch (my first tasks as intern 20+ years ago) are replaced with something like the following that anyone can do in 5 minutes

```
model = Sequential(Linear(N, L), ReLU(), Linear(L, D), Softmax())
opt = optimizer(model.parameters(), lr=0.001)
opt.fit(model, loss, data)
```

The key elements are automatic differentiation for derivatives and robust and easy-to-use optimization routines

Detour: Reverse-mode automatic differentiation

Assume a function $f(\boldsymbol{\theta}) : \mathbb{R}^D \to \mathbb{R}$ (think of loss functions or $\log p(\cdot)$)

Reverse-mode automatic differentiation allows computing all D partial derivatives of that function with small computational overhead

Split the function into simple expressions, store all intermediate results, and use derivatives of the simple expression to construct code that computes the derivative

Corresponds to backpropagation for neural networks, but works for general functions

Wikipedia has quite nice explanation and examples

Detour: Modern stochastic gradient descent

Now we know how to compute gradients of arbitrary loss functions $\mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda})$ with respect to the parameters $\boldsymbol{\lambda}$ of the model

Most losses are sums over individual data instances, such that $\mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}) = \frac{1}{n} \sum_{i=1}^{n} L(x_i, \boldsymbol{\lambda})$, and for optimization we would typically use stochastic gradient descent

After initialization we iterate for

1. Obtain stochastic estimate for the gradient based on a subset of the data instances (here just one): $\nabla_{\boldsymbol{\lambda}} \mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}_t) \approx \nabla_{\boldsymbol{\lambda}} L(x_i, \boldsymbol{\lambda}_t) =: g(\boldsymbol{\lambda})$
2. Update the parameters $\boldsymbol{\lambda}_{t+1} = \boldsymbol{\lambda}_t - \eta_t g(\boldsymbol{\lambda})$ using some step length η_t

During optimization η_t should decrease at suitable pace, typically adaptively (AdaGrad, Adam, RMSProp, ...)

Surprisingly tricky to implement well, but good robust solutions exist in all reasonable libraries

Model-independent variational approximation

Can we do VI for arbitrary models and approximations, without

(a) Making the mean-field assumption

(b) Assuming conjugate models

(c) ...or even knowledge of the model when writing the inference algorithm

Yes, but we do need some alternative assumptions. We need to assume either that

1. The approximation $q(\boldsymbol{\theta}|\boldsymbol{\lambda})$ is differentiable wrt to $\boldsymbol{\lambda}$
2. The model $p(\boldsymbol{x}, \boldsymbol{\theta})$ is differentiable wrt to $\boldsymbol{\theta}$ and the approximation can be reparameterized

Note that the former is very mild assumption (we get to choose the approximation, and pretty much all distributions you can think of have continuous parameterization), but the latter does not hold for models with discrete parameters θ

Model-independent variational approximation

The basic idea is fairly simple:

- Approximate the $\mathcal{L}_{ELBO}(\boldsymbol{\lambda})$ with a Monte Carlo estimate, computed by averaging the value over samples drawn from the approximation
- Use gradient descent to learn the parameters $\boldsymbol{\lambda}$ of the approximation, with $\boldsymbol{\lambda}_{t+1} = \boldsymbol{\lambda}_t + \alpha \nabla \mathcal{L}_{ELBO}(\boldsymbol{\lambda}_t)$

The approximation for ELBO can easily be written as objective for any deep learning framework and for optimization we can use standard routines with derivatives computed using automatic differentiation

However, we do not want to minimize the approximation for the ELBO but rather the true ELBO. We can estimate the gradients of that, but it is not trivial

Monte Carlo approximation for ELBO

Monte Carlo approximation for the bound itself is easy using

$$\mathcal{L}_{ELBO}(\boldsymbol{\lambda}) = \int q(\boldsymbol{\theta}|\boldsymbol{\lambda}) \left[\log p(\mathcal{D}, \boldsymbol{\theta}) - \log q(\boldsymbol{\theta}|\boldsymbol{\lambda})\right] d\boldsymbol{\theta} \approx \frac{1}{M} \sum_{m=1}^{M} \left[\log p(\mathcal{D}, \boldsymbol{\theta}_m) - \log q(\boldsymbol{\theta}_m|\boldsymbol{\lambda})\right],$$

where $\boldsymbol{\theta}_m$ are drawn from the approximation

From the perspective of a deep learning library this is a perfectly valid optimization objective, since it is simply a sum of logarithms evaluated for some parameter values: We always evaluate the loss for some particular data points and parameter values, and now only need to provide $\boldsymbol{\theta}_m$ as additional inputs

Derivatives of expectations

To compute the gradient of the objective we can safely change the order of integration (that is over $\boldsymbol{\theta}$) and differentiation (that is over $\boldsymbol{\lambda}$) to get

$$\begin{aligned}\nabla_{\boldsymbol{\lambda}} \mathcal{L}_{ELBO}(\boldsymbol{\lambda}) &= \nabla_{\boldsymbol{\lambda}} \int q(\boldsymbol{\theta}|\boldsymbol{\lambda}) \left[\log p(\mathcal{D}, \boldsymbol{\theta}) - \log q(\boldsymbol{\theta}|\boldsymbol{\lambda})\right] d\boldsymbol{\theta} \\ &= \int \nabla_{\boldsymbol{\lambda}} q(\boldsymbol{\theta}|\boldsymbol{\lambda}) \left[\log p(\mathcal{D}, \boldsymbol{\theta}) - \log q(\boldsymbol{\theta}|\boldsymbol{\lambda})\right] d\boldsymbol{\theta} - \int \nabla_{\boldsymbol{\lambda}} q(\boldsymbol{\theta}|\boldsymbol{\lambda}) \frac{q(\boldsymbol{\theta}|\boldsymbol{\lambda})}{q(\boldsymbol{\theta}|\boldsymbol{\lambda})} d\boldsymbol{\theta}\end{aligned}$$

where the last term disappears because of $\nabla_{\boldsymbol{\lambda}} \int q(\boldsymbol{\theta}|\boldsymbol{\lambda}) d\boldsymbol{\theta} = 0$.

Unfortunately this is not an expectation: We do not have integral of the form $\int q(\boldsymbol{\theta}|\boldsymbol{\lambda}) f(\boldsymbol{\theta}) d\boldsymbol{\theta}$ but the integrand is simply a product of some arbitrary terms, the gradient of the approximation and log-densities. Hence, we cannot use Monte Carlo approximation

However, there are two alternative strategies for computing the gradients: score function estimators and reparameterization

Score-function estimator

Chain-rule of differentiation gives

$$\nabla \log f(\boldsymbol{\theta}) = \frac{\nabla f(\boldsymbol{\theta})}{f(\boldsymbol{\theta})} \qquad \rightarrow \qquad \nabla f(\boldsymbol{\theta}) = f(\boldsymbol{\theta}) \nabla \log f(\boldsymbol{\theta})$$

If we denote $g(\boldsymbol{\theta}, \boldsymbol{\lambda}) = \log p(\mathcal{D}, \boldsymbol{\theta}) - \log q(\boldsymbol{\theta}|\boldsymbol{\lambda})$ then the derivative of the expectation is

$$\begin{aligned}\nabla_{\boldsymbol{\lambda}} \mathbb{E}_{q(\boldsymbol{\theta}|\boldsymbol{\lambda})} \left[g(\boldsymbol{\theta}, \boldsymbol{\lambda}) \right] &= \int g(\boldsymbol{\theta}, \boldsymbol{\lambda}) \nabla_{\boldsymbol{\lambda}} q(\boldsymbol{\theta}|\boldsymbol{\lambda}) d\boldsymbol{\theta} \\ &= \int g(\boldsymbol{\theta}, \boldsymbol{\lambda}) \left[q(\boldsymbol{\theta}|\boldsymbol{\lambda}) \nabla_{\boldsymbol{\lambda}} \log q(\boldsymbol{\theta}|\boldsymbol{\lambda}) \right] d\boldsymbol{\theta} = \mathbb{E}_{q(\boldsymbol{\theta}|\boldsymbol{\lambda})} \left[g(\boldsymbol{\theta}, \boldsymbol{\lambda}) \nabla_{\boldsymbol{\lambda}} \log q(\boldsymbol{\theta}|\boldsymbol{\lambda}) \right]\end{aligned}$$

which is an expectation again

Score-function estimator

This expectation is easy to approximate with Monte Carlo

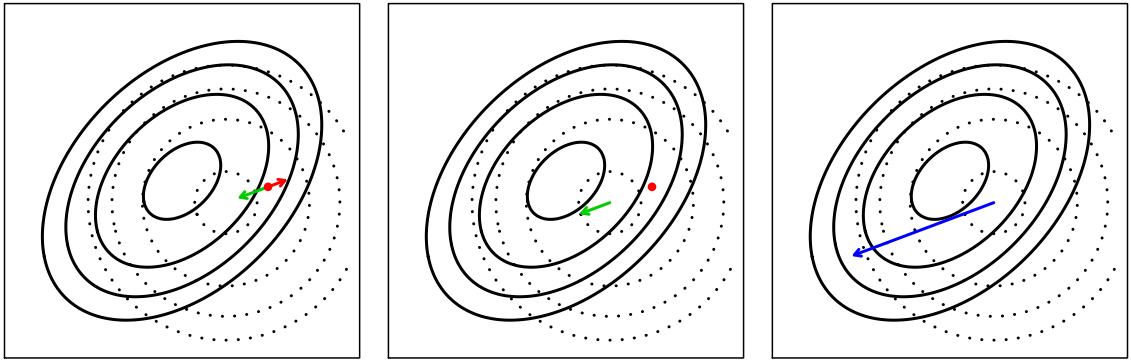
$$\nabla_{\boldsymbol{\lambda}} \mathbb{E}_{q(\boldsymbol{\theta}|\boldsymbol{\lambda})} \left[g(\boldsymbol{\theta}, \boldsymbol{\lambda}) \right] \approx \frac{1}{M} \sum_{m=1}^{M} g(\boldsymbol{\theta}_m, \boldsymbol{\lambda}) \nabla_{\boldsymbol{\lambda}} \log q(\boldsymbol{\theta}_m|\boldsymbol{\lambda}) = \frac{1}{M} \sum_{m=1}^{M} w_m \nabla_{\boldsymbol{\lambda}} \log q(\boldsymbol{\theta}_m|\boldsymbol{\lambda})$$

for $\boldsymbol{\theta}_m \sim q(\boldsymbol{\theta}|\boldsymbol{\lambda})$ and $w_m = g(\boldsymbol{\theta}_m, \boldsymbol{\lambda}) = \log p(\boldsymbol{x}, \boldsymbol{\theta}_m) - \log q(\boldsymbol{\theta}_m|\boldsymbol{\lambda})$

The estimator is unbiased, but unfortunately has high variance:

- The range of $\log p(\boldsymbol{x}, \boldsymbol{\theta}_m) - \log q(\boldsymbol{\theta}_m|\boldsymbol{\lambda})$ can be very large, so only the largest elements matter for the expectation
- The model itself does not directly contribute to the direction of the gradient in any way, but we are merely trying to move the approximation closer to each of the samples that were drawn from the approximation. The only information for pulling it towards the posterior comes from the weights

Score-function estimator

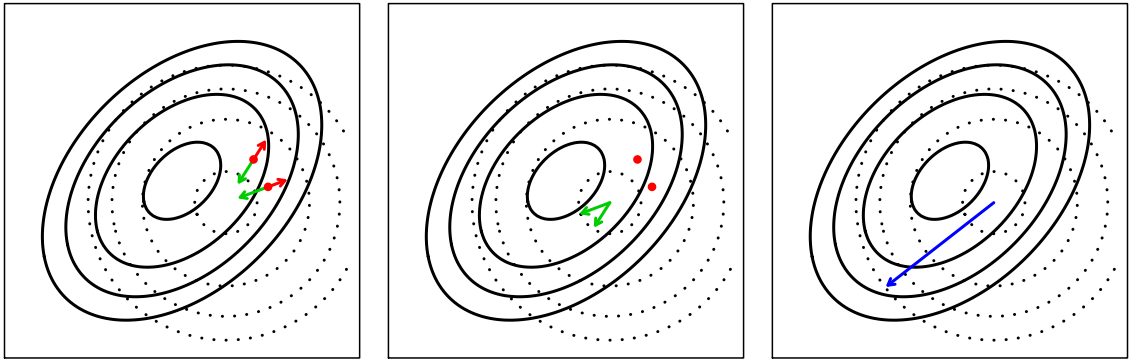


$\nabla_{\boldsymbol{\lambda}} \log q(\boldsymbol{\theta}_m|\boldsymbol{\lambda})$ $\log p(\boldsymbol{x}, \boldsymbol{\theta}_m) \nabla_{\boldsymbol{\lambda}} \log q(\boldsymbol{\theta}_m|\boldsymbol{\lambda})$

$\frac{1}{M} \sum_{m=1}^{M} \log p(\boldsymbol{x}, \boldsymbol{\theta}_m) \nabla_{\boldsymbol{\lambda}} \log q(\boldsymbol{\theta}_m|\boldsymbol{\lambda})$

Note: Scales altered for better visualization

Score-function estimator

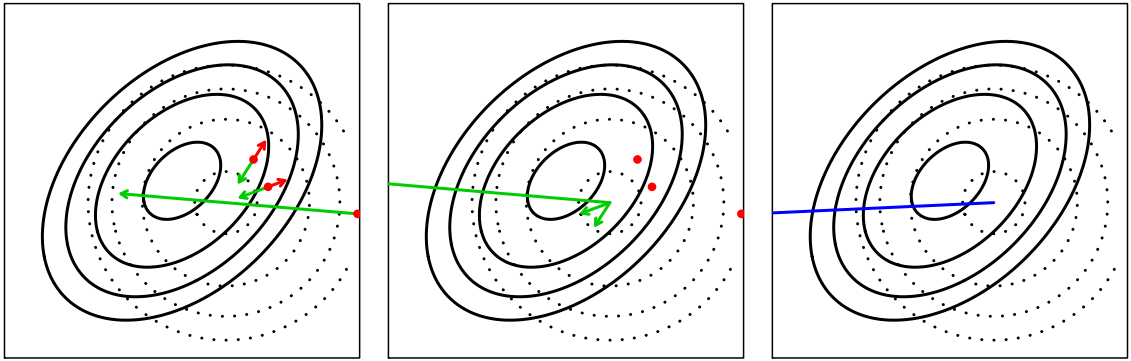


$\nabla_{\boldsymbol{\lambda}} \log q(\boldsymbol{\theta}_m|\boldsymbol{\lambda})$ $\log p(\boldsymbol{x}, \boldsymbol{\theta}_m) \nabla_{\boldsymbol{\lambda}} \log q(\boldsymbol{\theta}_m|\boldsymbol{\lambda})$

$\frac{1}{M} \sum_{m=1}^{M} \log p(\boldsymbol{x}, \boldsymbol{\theta}_m) \nabla_{\boldsymbol{\lambda}} \log q(\boldsymbol{\theta}_m|\boldsymbol{\lambda})$

Note: Scales altered for better visualization

Score-function estimator

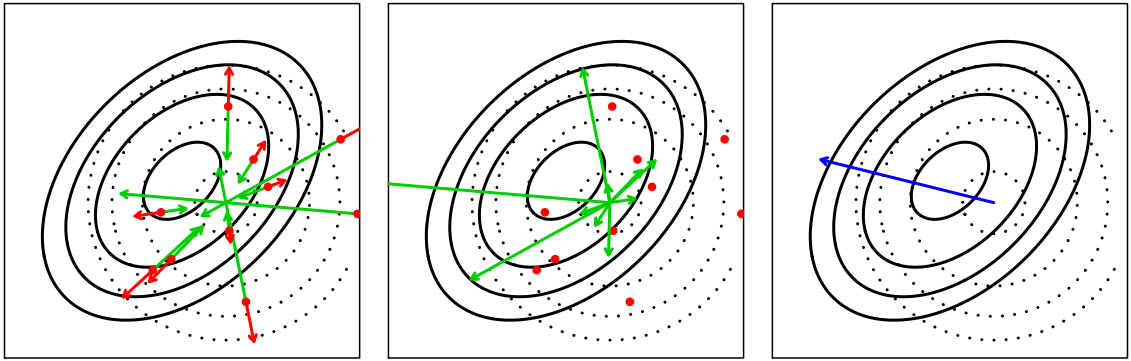


$\nabla_{\boldsymbol{\lambda}} \log q(\boldsymbol{\theta}_m|\boldsymbol{\lambda})$ $\log p(\boldsymbol{x}, \boldsymbol{\theta}_m) \nabla_{\boldsymbol{\lambda}} \log q(\boldsymbol{\theta}_m|\boldsymbol{\lambda})$

$\frac{1}{M} \sum_{m=1}^{M} \log p(\boldsymbol{x}, \boldsymbol{\theta}_m) \nabla_{\boldsymbol{\lambda}} \log q(\boldsymbol{\theta}_m|\boldsymbol{\lambda})$

Note: Scales altered for better visualization

Score-function estimator

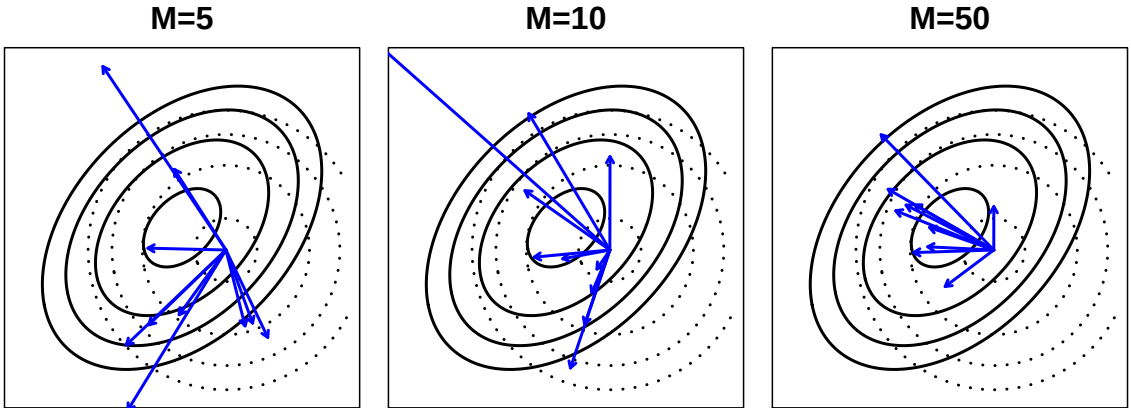


$\nabla_{\boldsymbol{\lambda}} \log q(\boldsymbol{\theta}_m|\boldsymbol{\lambda})$ $\log p(\boldsymbol{x}, \boldsymbol{\theta}_m) \nabla_{\boldsymbol{\lambda}} \log q(\boldsymbol{\theta}_m|\boldsymbol{\lambda})$

$\frac{1}{M} \sum_{m=1}^{M} \log p(\boldsymbol{x}, \boldsymbol{\theta}_m) \nabla_{\boldsymbol{\lambda}} \log q(\boldsymbol{\theta}_m|\boldsymbol{\lambda})$

Note: Scales altered for better visualization

Score-function estimator



$\frac{1}{M}\sum_{m=1}^{M} \log p(\boldsymbol{x}, \boldsymbol{\theta}_m) \nabla_{\boldsymbol{\lambda}} \log q(\boldsymbol{\theta}_m | \boldsymbol{\lambda})$

Note: Scales altered for better visualization

Score-function estimator

For practical VI algorithms we need either very large M or other means for reducing the variance

Control variates: Assuming we already know $\mathbb{E}[h(X)]$ for some function $h(\cdot)$, we can compute

$$\mathbb{E}[f(X)] = \mathbb{E}[f(X) - h(X)] + \mathbb{E}[h(X)]$$

due to linearity of expectation. The variance of the first term is

$$\mathrm{Var}(f - h) = \mathrm{Var}(f) - 2\mathrm{Cov}(f, h) + \mathrm{Var}(h)$$

which is smaller than $\mathrm{Var}(f)$ for positively correlated functions (and exactly zero if $f = h$)

Variance reduction

Apply the control variate idea by using $h(X) = C\nabla_{\boldsymbol{\lambda}} \log q(\boldsymbol{\theta}|\boldsymbol{\lambda})$, resulting in

$$\mathbb{E}_{q(\boldsymbol{\theta}|\boldsymbol{\lambda})}\left[g(\boldsymbol{\theta},\boldsymbol{\lambda})\nabla_{\boldsymbol{\lambda}} \log q(\boldsymbol{\theta}|\boldsymbol{\lambda}) - C\nabla_{\boldsymbol{\lambda}} \log q(\boldsymbol{\theta}|\boldsymbol{\lambda})\right] + \mathbb{E}_{q(\boldsymbol{\theta}|\boldsymbol{\lambda})}\left[C\nabla_{\boldsymbol{\lambda}} \log q(\boldsymbol{\theta}|\boldsymbol{\lambda})\right]$$

By again applying the log-derivative identity, the last term simplifies to

$$C\mathbb{E}_{q(\boldsymbol{\theta}|\boldsymbol{\lambda})}\left[\frac{\nabla_{\boldsymbol{\lambda}} q(\boldsymbol{\theta}|\boldsymbol{\lambda})}{q(\boldsymbol{\theta}|\boldsymbol{\lambda})}\right] = C\int \nabla_{\boldsymbol{\lambda}} q(\boldsymbol{\theta}|\boldsymbol{\lambda}) d\boldsymbol{\theta} = C\nabla_{\boldsymbol{\lambda}} \int q(\boldsymbol{\theta}|\boldsymbol{\lambda}) d\boldsymbol{\theta} = 0$$

because an integral of any distribution is one by definition and hence constant wrt to $\boldsymbol{\lambda}$

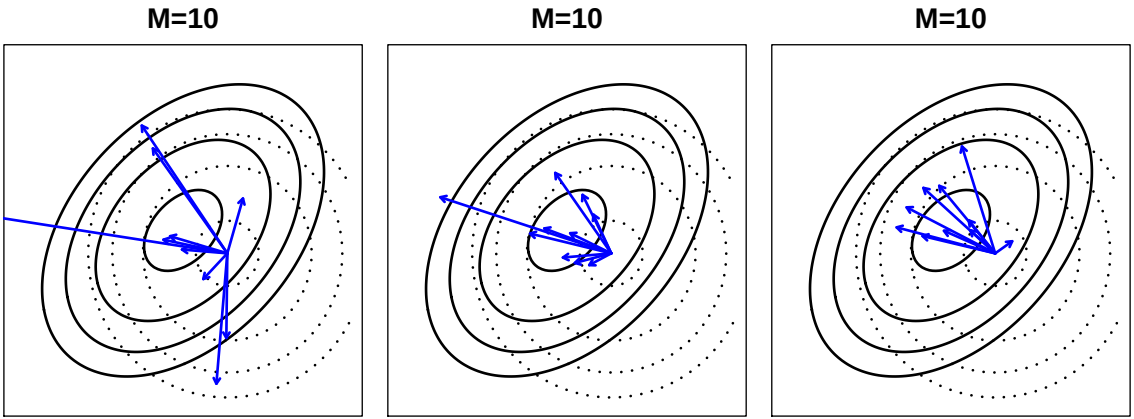
Variance reduction

Having got rid of the last term, the estimator becomes

$$\begin{aligned}&\mathbb{E}_{q(\boldsymbol{\theta}|\boldsymbol{\lambda})}\left[g(\boldsymbol{\theta},\boldsymbol{\lambda})\nabla_{\boldsymbol{\lambda}}\log q(\boldsymbol{\theta}|\boldsymbol{\lambda}) - C\nabla_{\boldsymbol{\lambda}}\log q(\boldsymbol{\theta}|\boldsymbol{\lambda})\right]\\&= \mathbb{E}_{q(\boldsymbol{\theta}|\boldsymbol{\lambda})}\left[(g(\boldsymbol{\theta},\boldsymbol{\lambda}) - C)\nabla_{\boldsymbol{\lambda}}\log q(\boldsymbol{\theta}|\boldsymbol{\lambda})\right],\end{aligned}$$

where we can choose C however we want to, in an attempt to reduce the variance – the equation is correct for all choices, including ones that depend on the data instance or other external factors even though the derivation was here done for a constant

Score-function estimator: Variance reduction with control variates



Control variate $C\nabla_{\boldsymbol{\lambda}} \log q(\boldsymbol{\theta}|\boldsymbol{\lambda})$ for $C = 0, C = -3,$ and $C = -6$

Reparameterization

The other main strategy for estimating gradients of ELBO builds on the idea of reparameterizing the approximation [Titsias and Lázaro-Gredilla, 2014, Kingma and Welling, 2014, Rezende et al., 2014]

Assume $q(\boldsymbol{\theta}|\boldsymbol{\lambda})$ can be written as a system

$$\mathbf{z} \sim \phi(\mathbf{z})$$
$$\boldsymbol{\theta} = f(\mathbf{z}, \boldsymbol{\lambda})$$

where $\phi(\mathbf{z})$ is some simple distribution that does not depend on $\boldsymbol{\lambda}$ and $f(\mathbf{z}, \boldsymbol{\lambda})$ is a deterministic transformation

The common example is $x \sim \mathcal{N}(\mu, \sigma)$ re-written using

$$z \sim \mathcal{N}(0, 1)$$
$$x = \mu + \sigma z$$

Reparameterization

With such representation we can re-write the expectation over $\phi(\mathbf{z})$ instead, using

$$\mathcal{L} = \mathbb{E}_{q(\boldsymbol{\theta}|\boldsymbol{\lambda})}\left[g(\boldsymbol{\theta}, \boldsymbol{\lambda})\right] = \mathbb{E}_{\phi(\mathbf{z})}\left[g(f(\mathbf{z}, \boldsymbol{\lambda}), \boldsymbol{\lambda})\right] = \mathbb{E}_{\phi(\mathbf{z})}\left[\log p(\mathcal{D}, f(\mathbf{z}, \boldsymbol{\lambda})) - \log q(f(\mathbf{z}, \boldsymbol{\lambda}), \boldsymbol{\lambda})\right]$$

where the expectation is now over a distribution with no parameters, and the original parameters of our approximation only appear in the transformation

Now we can easily push the derivative inside the expectation and use standard Monte Carlo

$$\nabla_{\boldsymbol{\lambda}}\mathcal{L} = \mathbb{E}_{\phi(\mathbf{z})}\left[\nabla_{\boldsymbol{\lambda}} g(f(\mathbf{z}, \boldsymbol{\lambda}), \boldsymbol{\lambda})\right] \approx \frac{1}{M}\sum_{m=1}^{M} \nabla_{\boldsymbol{\lambda}} g(f(\mathbf{z}_m, \boldsymbol{\lambda}), \boldsymbol{\lambda})$$

for $\mathbf{z}_m \sim \phi(\mathbf{z})$

Reparameterization

Suddenly we are computing derivatives of the actual model, since

$$\nabla_{\boldsymbol{\lambda}} \log p(\boldsymbol{x}, f(\mathbf{z}, \boldsymbol{\lambda})) = \nabla_{\theta} \log p(\boldsymbol{x}, \theta) \nabla_{\boldsymbol{\lambda}} f(\mathbf{z}, \boldsymbol{\lambda})$$

This is good news because now the model influences the gradient, but also bad news in the sense that we need to be able to compute derivatives of the model – this rules out models with discrete parameters

Reparameterization

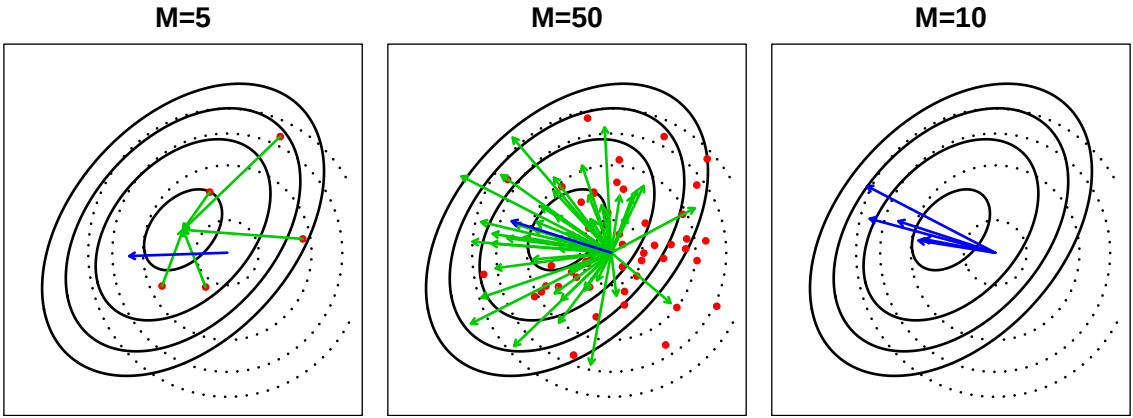
Simplest example:

$$q(\theta|\mu,\sigma) = \mathcal{N}(\mu,\sigma^2) \qquad \equiv \qquad z \sim \mathcal{N}(0,1), \quad \theta = \mu + \sigma z$$

The gradient for the log-density part of ELBO becomes

$$\frac{\partial \log p(\mathcal{D},\boldsymbol{\theta})}{\partial \mu} = \frac{\partial \log p(\mathcal{D},\boldsymbol{\theta})}{\partial \theta}\frac{\partial f(z,\boldsymbol{\lambda})}{\partial \mu} = \frac{\partial \log p(\mathcal{D},\boldsymbol{\theta})}{\partial \theta}$$
$$\frac{\partial \log p(\mathcal{D},\boldsymbol{\theta})}{\partial \sigma} = \frac{\partial \log p(\mathcal{D},\boldsymbol{\theta})}{\partial \theta}\frac{\partial f(z,\boldsymbol{\lambda})}{\partial \sigma} = \frac{\partial \log p(\mathcal{D},\boldsymbol{\theta})}{\partial \theta} z$$

Reparameterization



Reparameterization

Reparameterization can be done for:

- One-liner samplers (normal, exponential, cauchy, ...)
- Rejection samplers (gamma, beta, dirichlet, ...)
- Chains of transformations (log-normal, inv-gamma)
- Implicit reparameterization [Figurnov et al., 2018]

Example of a chain: $z \sim \mathcal{N}(0,1) \quad t = \mu + \sigma z \quad \theta = e^t$

Target	$p(z;\theta)$	Base $p(\epsilon)$	One-liner $g(\epsilon;\theta)$
Exponential	$\exp(-x); x > 0$	$\epsilon \sim [0;1]$	$\ln(1/\epsilon)$
Cauchy	$\frac{1}{\pi(1+x^2)}$	$\epsilon \sim [0;1]$	$\tan(\pi\epsilon)$
Laplace	$\mathcal{L}(0;1) = \exp(-\lvert x\rvert)$	$\epsilon \sim [0;1]$	$\ln(\frac{\epsilon_1}{\epsilon_2})$
Laplace	$\mathcal{L}(\mu;b)$	$\epsilon \sim [0;1]$	$\mu - bsgn(\epsilon)\ln(1-2\lvert\epsilon\rvert)$
Std Gaussian	$\mathcal{N}(0;1)$	$\epsilon \sim [0;1]$	$\sqrt{\ln(\frac{1}{\epsilon_1})}\cos(2\pi\epsilon_2)$
Gaussian	$\mathcal{N}(\mu;RR^\top)$	$\epsilon \sim \mathcal{N}(0;1)$	$\mu + R\epsilon$
Rademacher	$Rad(\frac{1}{2})$	$\epsilon \sim Bern(\frac{1}{2})$	$2\epsilon - 1$
Log-Normal	$\ln\mathcal{N}(\mu;\sigma)$	$\epsilon \sim \mathcal{N}(\mu;\sigma^2)$	$\exp(\epsilon)$
Inv Gamma	$i\mathcal{G}(k;\theta)$	$\epsilon \sim \mathcal{G}(k;\theta^{-1})$	$\frac{1}{\epsilon}$

Table from `http://blog.shakirm.com/2015/10/machine-learning-trick-of-the-day-4-reparameterisation-tricks/`

Score function vs reparameterization

Score function: Gradients point towards the mode of the approximation, and the only way the model influences them is through $\log p(\mathcal{D}, \boldsymbol{\theta}_m)$ in the weights

Reparameterization: Gradients point towards the mode of the posterior

The latter is clearly better in case $\log p(\mathcal{D}, \boldsymbol{\theta})$ is differentiable and we can reparameterize the approximation, but the former is more general

Often for reparameterization gradients even $M = 1$ is enough, whereas score function estimator requires orders of magnitude more even with good control variates

Automatic variational inference

The above derivations lead to practical algorithms for model-free variational inference

- Assume a differentiable (and usually still factorized) approximation $q(\boldsymbol{\theta}|\boldsymbol{\lambda})$
- Black-box VI [Ranganath et al., 2014]: If nothing can be assumed about $\log p(\boldsymbol{\theta}, \boldsymbol{\lambda})$ then use the score fuction estimator

$$\mathbb{E}_{q(\boldsymbol{\theta}|\boldsymbol{\lambda})}\left[(g(\boldsymbol{\theta}, \boldsymbol{\lambda}) - C(\boldsymbol{x}))\nabla_{\boldsymbol{\lambda}} \log q(\boldsymbol{\theta}|\boldsymbol{\lambda})\right]$$

  with suitable control variate
- Reparameterization VI [Titsias and Lázaro-Gredilla, 2014]: If $\log p(\boldsymbol{\theta}, \boldsymbol{\lambda})$ is differentiable and $q(\boldsymbol{\theta}|\boldsymbol{\lambda})$ can be reparameterized, use the reparameterization estimate

$$\mathbb{E}_{\phi}\left[\nabla_{\boldsymbol{\lambda}} g(f(\mathbf{z}, \boldsymbol{\lambda}), \boldsymbol{\lambda})\right]$$

Automatic variational inference

Irrespective of the estimator, the actual learning is done with stochastic gradient ascent, in a similar manner as we would do for deep learning

- Use automatic differentiation for computing the required gradients
- You can safely use mini-batches for evaluating the gradient – it is stochastic anyway due to finite M, so no point in looping over all data points either
- Update the parameters $\boldsymbol{\lambda}$ according to the gradient, using your favorite optimization algorithm

Alternative by transforming parameters

An alternative is to transform the model parameters $\boldsymbol{\theta}$ into $\hat{\boldsymbol{\theta}}$ so that $q(\hat{\theta}_d) = \mathcal{N}(\hat{\theta}_d)$ is a reasonable approximation for every factor. We can then always use reparameterization for normal distribution.

For example, for $\theta \geq 0$ we can use $\hat{\theta} = \log(e^{\theta} - 1)$ to map to the whole real line and then approximate the posterior with $q(\hat{\theta})$

Due to the transformation we need to add the log-Jacobian of the backwards transformation $\theta = e^{\hat{\theta}} + 1$ when evaluating the gradient, but this can also be computed with automatic differentiation

This is how Stan does VI [Kucukelbir et al., 2017] because they anyway need to map the model parameters to unconstrained space for HMC

Gradient-based VI in practice: PPCA

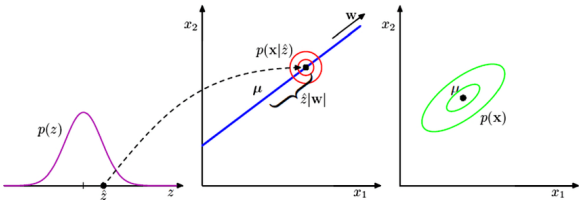
To better grasp how this works in practice, let us consider a simple case of probabilistic PCA [Bishop, 1999], simplified so that we only do Bayesian inference over $\mathbf{x}$ and not $\boldsymbol{W}$. For example Ilin and Raiko [2010] provide CAVI updates for the full model

Model:

$$\begin{aligned}\boldsymbol{x}_j &\sim \mathcal{N}(0, I)\\ \epsilon_j &\sim \mathcal{N}(0, \sigma^2)\\ \mathbf{y}_j &= \boldsymbol{W}\boldsymbol{x}_j + \epsilon_j\end{aligned}$$

Approximation:

$$\begin{aligned}q(\mathbf{X}|\boldsymbol{\lambda}) &= \prod_j q(\mathbf{x}_j|\boldsymbol{\lambda}_j),\\ q(\mathbf{x}_j|\boldsymbol{\lambda}_j) &= \mathcal{N}(\bar{\boldsymbol{x}}_j, \bar{\boldsymbol{\Sigma}}_j)\end{aligned}$$



Gradient-based VI in practice: PPCA

For CAVI the choice of normal approximation followed directly from the mean-field assumption and the model, but here we had to assume it

Since the approximations are multivariate normal, we use the reparameterization

$$\boldsymbol{z} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I}), \qquad \boldsymbol{x}_j = \bar{\boldsymbol{x}}_j + \boldsymbol{R}_j \boldsymbol{z},$$

where $\boldsymbol{R}$ is the Cholesky decomposition of the covariance matrix so that $\bar{\boldsymbol{\Sigma}}_j = \boldsymbol{R}\boldsymbol{R}^T$. That is, we actually parameterize each term with $\bar{\boldsymbol{x}}_j$ and $\boldsymbol{R}_j$

Now a SGD implementation requires only:

- Specifying the collection of n approximation terms with parameters $\bar{\boldsymbol{x}}_j$ and $\boldsymbol{R}_j$
- Function that can evaluate $\log p(\boldsymbol{Y}, \boldsymbol{X_m}) - \log q(\boldsymbol{X_m}, \boldsymbol{\lambda})$ for any $\boldsymbol{X}_m$ sampled from the approximation
- Function that evaluates the gradient using the reparameterization estimate

Gradient-based VI in practice: PPCA

Gradient-based optimization is here going to CAVI, but has one significant advantage: We can change the model almost arbitrarily without needing to change the inference code

Changing the likelihood and prior only requires modifying the log-density $\log p(\boldsymbol{\theta}, \mathcal{D})$; things like robust PCA that replaces the normal likelihood with a distribution with heavier tails (e.g. student-t) is trivial, as would be a model that encourages sparse latent representations

Furthermore, we can change the model itself, relaxing core assumptions like the linear dependency between the latent representations $\boldsymbol{x}_j$ and the observations $\boldsymbol{y}_j$

Nonlinear PPCA

For every sample, the PPCA model assumes the vector of observations is linear function of the latent representation: $\mathbf{y}_j \approx \boldsymbol{W}\boldsymbol{x}_j$

To get a richer model we can make this non-linear by plugging in some neural network instead: $\mathbf{y}_j \approx d(\boldsymbol{x}_j, \boldsymbol{\alpha}_d)$, where $\boldsymbol{\alpha}_d$ are the parameters of the network

We can still do variational inference in the same way:

- If we still assume a normal likelihood we do not need to change $\log p(\boldsymbol{\theta}, \boldsymbol{\lambda})$ at all, we just feed in $d(\boldsymbol{x}_j, \boldsymbol{\alpha}_d)$ in place of $\boldsymbol{W}\boldsymbol{x}_j$
- We still have separate variational approximation term $q(\boldsymbol{x}_j|\bar{\boldsymbol{x}}_j, \boldsymbol{R}_j)$ for each latent representation

Nonlinear PPCA

The parameters $\boldsymbol{\alpha}_d$ that define the function are treated as hyperparameters, rather than random variables, similar to how we have been treating $\boldsymbol{W}$ in this simplified model

In terms of software implementation, we simply define them as optimizable parameters, so that the software library is optimizing over $\{\bar{\boldsymbol{x}}_j, \boldsymbol{R}_j\}_j^n$ and $\boldsymbol{\alpha}_d$ jointly

No need to do anything else: Since the objective and the gradient depend on $\boldsymbol{\alpha}_d$, the automatic differentiation engine will anyway end up computing the derivative also with respect to these

In summary: We can change the model itself almost arbitrarily, without needing to change the derivations

How about the approximation?

While changing the model, we have been keeping the approximation fixed: For every term we still have $\bar{\boldsymbol{x}}_j$ and $\boldsymbol{R}_j$ as independent free parameters. This may be a problem for large n

The CAVI updates for these parameters (simplified from Ilin and Raiko [2010]) are actually functions of the observed data:

$$\boldsymbol{\Sigma}_j = \nu_y \left(\nu_y \boldsymbol{I} + \boldsymbol{W}\boldsymbol{W}^T \right)^{-1}$$
$$\bar{\boldsymbol{x}}_j = \frac{1}{\nu_y} \boldsymbol{\Sigma}_j \boldsymbol{W} \boldsymbol{y}_j$$

They depend on some unknown parameters (the current $\boldsymbol{W}$), but still we observe that

- The optimal parameters for each $q(\boldsymbol{x}_j)$ are some functions of the observed features $\boldsymbol{y}_j$ of that particular data sample
- The functions are quite simple, only consisting of matrix products and inverses

How about the approximation?

Building on this intuition, let's change the way we parameterize the approximation

Rather than assuming each approximation term has independent parameterization, let's define flexible parametric models (neural networks) that map the observed data to the parameters of the approximation:

$$\bar{\boldsymbol{x}}_j = e_\mu(\boldsymbol{y}_j, \boldsymbol{\alpha}_{e_\mu})$$
$$\boldsymbol{R}_j = e_R(\boldsymbol{y}_j, \boldsymbol{\alpha}_{e_R})$$

Here we have two separate networks since the former needs to output c-dimensional vectors and the latter vectorized Cholesky decompositions, but the early layers of these networks would often be shared

Amortized inference

Variational inference replacing free approximation parameters for every latent variable with a generic function providing the parameters is called amortized inference[1]

Previously we had n parameter vectors of c dimensions and n vectors of $c(c+1)/2$ dimensions, whereas now we have only fixed-dimensional vectors $\boldsymbol{\alpha}_{e_\mu}$ and $\boldsymbol{\alpha}_{e_R}$

Network parameters again treated as hyperparameters: We simply replace all of $\bar{\boldsymbol{x}}_j$ as optimizable parameters with $\boldsymbol{\alpha}_{e_\mu}$ (and analogously for $\boldsymbol{R}$) but do not need to change anything else – the automatic differentiation engine takes care of propagating the gradients

[1] Wikipedia: *Amortization is paying off an amount owed over time by making planned, incremental payments of principal and interest. To amortise a loan means "to kill it off". In accounting, amortisation refers to charging or writing off an intangible asset's cost as an operational expense over its estimated useful life to reduce a company's taxable income.*

Amortized inference

Properties:

- For large n we will have way less parameters, which speeds up training
- For simple enough models the optimal parameters are fairly simple functions of data, so we can probably still learn the same solution
- Regularization effect of some kind: For example, we now have identical parameters for identical observation vectors, which would be difficult to obtain with direct optimization
- Note that we did not change the approximation itself, just its parameterization: All approximation terms are still multivariate normal

Amortized inference for the example model

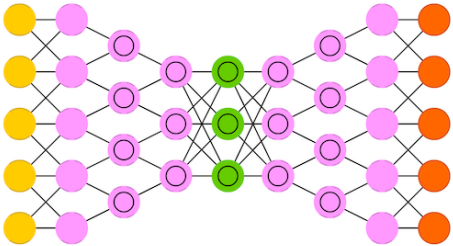
We started with the simple linear model model and ended up proposing three ways of modifying the model to be more flexible:

- Change of distributions: We can freely change the priors and the likelihood
- Non-linear modeling: Replace linear mapping from latent variables to observations with non-linear decoder $d(\boldsymbol{x}_j, \boldsymbol{\alpha}_d)$
- Amortized inference: Replace free parameters for the approximation terms with non-linear encoder $e(\boldsymbol{y}_j, \boldsymbol{\alpha}_e)$

Variational autoencoder

This general model is called variational autoencoder (VAE) [Kingma and Welling, 2014] in deep learning literature, because of the way the computation resembles the autoencoding principle

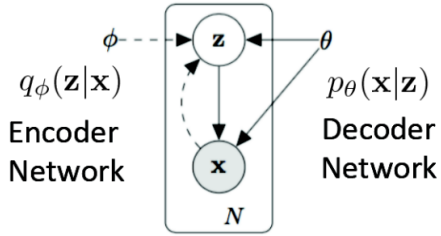
An autoencoder is model that feeds inputs $\boldsymbol{x}$ through sequence of encoder and decoder networks and is trained to minimize a reconstruction error, in an attempt to learn a low-dimensional representation (code) for the inputs that captures the structure of the data



Variational autoencoder

VAE looks pretty much identical to a standard autoencoder and the literature uses the same terminology for both, but it is not an autoencoder in the same sense:

- We do not actually have an encoder that processes data samples, but we merely use the encoder network to learn parameters of a posterior approximation
- The decoder outputs parameters of a probabilistic model (likelihood) for the observed data, not reconstructions of the observations



To help making the distinction, it is better to draw VAE so that it does not resemble autoencoders visually

Variational autoencoder

Variational autoencoder as Bayesian model:

- Model:

$$\begin{aligned}\boldsymbol{x}_j &\sim \mathcal{N}(0, \boldsymbol{I}) \\ \boldsymbol{\epsilon}_j &\sim \mathcal{N}(0, \sigma^2 \boldsymbol{I}) \\ \boldsymbol{y}_j &= d(\boldsymbol{x}_j, \boldsymbol{\alpha}_d) + \boldsymbol{\epsilon}_j\end{aligned}$$

- Approximation:

$$q(\boldsymbol{X}) = \prod_j^n q(\mathbf{x}_j) = \prod_j^n \mathcal{N}(e_\mu(\mathbf{y}_j, \boldsymbol{\alpha}_{e_\mu}), \mathrm{diag}(e_\sigma(\boldsymbol{y}_j, \alpha_{e_\sigma})))$$

Typically the approximation for $q(\mathbf{x}_j)$ has diagonal covariance, and hence both $e_\mu(\cdot)$ and $e_\sigma(\cdot)$ output vectors of c elements, but the latter has to output positive values to provide valid covariance.

Variational autoencoder

VAE is actually quite simple model that uses a simple approximation:

- We only do Bayesian inference over $\boldsymbol{x}_j$ and not over any other variables of the model
- The model is essentially just Bayesian PCA that replaces the linear mapping with a neural network
- We use (almost always) factorized normal approximation for the posterior, and typically also a normal prior for $\boldsymbol{x}_j$

Normalizing flows

An important detail about VAE – or more generally about amortized inference – is that the approximation itself is not changed, only its parameterization. If anything, we are constraining the approximation by replacing the free parameters with some functions from the input data

Amortized inference uses the reparameterization: $\boldsymbol{\theta} = e_{\mu}(\boldsymbol{x}_j) + e_{\sigma}(\boldsymbol{x}_j)\boldsymbol{z}$

One way to get richer approximations is to feed in the random sample $\boldsymbol{z} \sim \phi(\boldsymbol{z})$ itself through a neural network $f(\boldsymbol{z}, \boldsymbol{\alpha})$ instead, using $\boldsymbol{\theta} = f(\boldsymbol{z})$. Then the density itself changes to

$$q(\boldsymbol{\theta}) = \phi(\boldsymbol{z}) \left| \det \frac{\partial f^{-1}}{\partial \boldsymbol{\theta}} \right| = \phi(\boldsymbol{z}) \left| \det \frac{\partial f}{\partial \boldsymbol{z}} \right|^{-1}$$

where $f^{-1}(\cdot)$ is the inverse transformation and the latter term is the determinant of its Jacobian

Normalizing flows

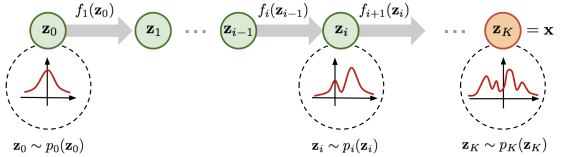
Now we can change the distribution itself, but need a network $f(\boldsymbol{z})$ that is invertible and for which we can compute the determinant of the Jacobian efficiently

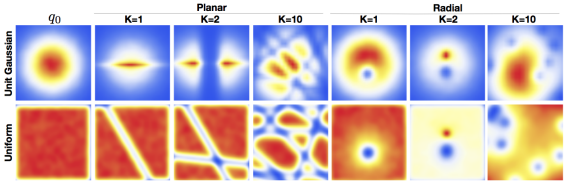
Normalizing flows [Rezende and Mohamed, 2015] are particular forms of neural network architectures designed to make this easy, by constructing the network by stacking layers with easy Jacobians

For example, a planar flow $f(\boldsymbol{z}) = \boldsymbol{z} + \boldsymbol{u}h(\boldsymbol{w}^T\boldsymbol{z} + b)$ with some scalar non-linearity $h(\cdot)$ has the Jacobian $|1 + \boldsymbol{u}^\phi(\boldsymbol{z})|$. Despite the simplicity, we can make more complex transformations by stacking multiple such layers

Normalizing flows can be used both as models for the data itself, or as more flexible variational approximations

Normalizing flows





Images from

`https://lilianweng.github.io/lil-log/2018/10/13/flow-based-deep-generative-models`

and `http://akosiorek.github.io/ml/2018/04/03/norm_flows.html`

Take home messages

- Variational inference over arbitrary probabilistic programs is possible, but besides model we need to somehow specify the approximation as well
- Deriving/understanding the gradient estimators is not particularly easy, but the implementation is fairly straightforward
- Can be done with the same basic tools as deep learning, since all we need is automatic differentiation, sampling from standard distributions, and SGD
- Consequently easy to merge with neural networks to get non-linear elements, but typically we still use point estimates over the network parameters
- This comes with a cost: Analytic integrals and exact coordinate ascent updates are both way faster and more reliable than Monte Carlo estimates and stochastic gradient optimization
- In conclusion: If CAVI is possible, use it. If not, do not try to extend the classical approach but rather switched to stochastic optimization

References I

Christopher M Bishop. Bayesian pca. *Advances in neural information processing systems*, 1999.

Mikhail Figurnov, Shakir Mohamed, and Andriy Mnih. Implicit Reparameterization Gradients. In *Advances in Neural Information Processing Systems 31*, 2018.

Alexander Ilin and Tapani Raiko. Practical approaches to principal component analysis in the presence of missing values. *Journal of Machine Learning Research*, 2010.

Michael I. Jordan, Zoubin Ghahramani, Tommi S. Jaakkola, and Lawrence K. Saul. An introduction to variational methods for graphical models. *Machine Learning*, 37(2):183–233, 1999.

D.P. Kingma and M. Welling. Auto-encoding variational Bayes. In *Proceedings of the 2nd International Conference on Learning Representations*, 2014.

Arto Klami. Polya-gamma augmentations for factor models. In *Proceedings of the Asian Conference on Machine Learning*, 2014.

Alp Kucukelbir, Dustin Tran, Rajesh Ranganath, Andrew Gelman, and David M Blei. Automatic Differentiation Variational Inference. *The Journal of Machine Learning Research*, 18(1), 2017.

References II

Nicholas G. Polson, James G. Scott, and Jess Windle. Bayesian inference for logistic models using polya-gamma latent variables. *Journal of the American Statistical Association*, 108(504):1339–1349, 2013.

Rajesh Ranganath, Sean Gerrish, and David Blei. Black Box Variational Inference. In *Proceedings of the 17th International Conference on Artificial Intelligence and Statistics*, 2014.

Danilo Rezende and Shakir Mohamed. Variational Inference with Normalizing Flow. In *Proceedings of the 32nd International Conference on Machine Learning*, 2015.

Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. Stochastic backpropagation and approximate inference in deep generative models. In *Proceedings of the 31st International Conference on Machine Learning*, 2014.

Matthias Seeger and Guillaume Bouchard. Fast variational Bayesian inference for non-conjugate matrix factorization models. In *Proceedings of AISTATS*, 2012.

Michalis Titsias and Miguel Lázaro-Gredilla. Doubly Stochastic Variational Bayes for non-Conjugate Inference. In *Proceedings of 31st International Conference on Machine Learning*, 2014.