Deep generative models

A deep generative model is one where a latent variable is first sampled from a simple distribution (e.g., an isotropic Gaussian) and then passed through a complex function (e.g., a multilayer perception (MLP)). Here is the most common formulation:

$p_\theta(x_i, z_i)$ **complete data likelihood** for data point i

$x_i \in \mathbb{R}^d$ **data point** (e.g., image, video, genomic sequence, ...)

$z_i \in \mathbb{R}^k$ **continuous latent variable**

θ **parameters** of the model

Generative process:

$z_i \sim \mathcal{N}(0, I)$ **sample latent** from standard multivariate normal

$x_i \sim \mathcal{N}(g_\theta(z_i), I)$ **sample data** from multivariate normal with mean equal to non-linear function of z_i

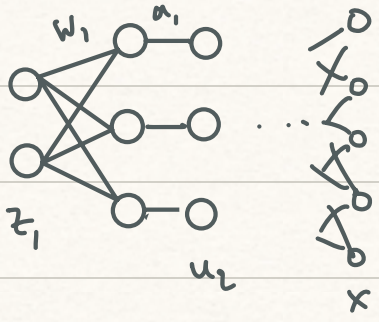
The non-linear function in this case will be a **multilayer perception (MLP)** aka feed-forward neural network which is defined by sequences of linear units and non-linear activation functions:

$$g_\theta(z_i) = g_L(g_{L-1}(\cdots g_1(z)\cdots))$$

Each successive vector is called a **hidden unit**, which is defined recursively:

$$u_{l+1} = g_l(u_l) = a_l(W_l u_l + b_l)$$

u_{l+1}: hidden unit; a_l: activation function; W_l: weights; b_l: bias



The activation function is applied element-wise to the input vector. A typical activation function is **RELU**:

$$a_l(c) \overset{e.g.}{=} \text{RELU}(a) = \max(0, a)$$

The parameters are thus just the weights and biases:

$$\theta = \{W_l, b_l\}_{l=1}^{L}$$

The more layers, the more expressive the model. Very complex distributions (e.g., the distributions of all images) can be modeled (in principle) with a deep generative model.

Inference and learning

We have two goals: fitting the parameters (learning) and using the fitted model to process data (inference).

learning $\underset{\theta}{\operatorname{argmax}}\ p_\theta(x) = \underset{\theta}{\operatorname{argmax}} \prod_{i=1}^{n} \int p_\theta(x_i, z_i)\, dz_i$

inference $p_\theta(z_i \mid x_i)$

Both are intractable, because we cannot marginalize z out; the relationship between z and x is too complex.

Evidence lower bound

As with other intractable models, we can set up a lower bound to the evidence:

$$\log p_\theta(x) \geq \underbrace{\mathbb{E}_{q(z)}\left[\log p_\theta(x,z) - \log q(z)\right]}_{\equiv B(q,\theta)\ \text{ELBO}}$$

where q(Z) is any surrogate or variational distribution over the latent variables. Assume that q factorizes, so:

$$= \sum_{i=1}^{n} \underbrace{\mathbb{E}_{q_i(z_i)}\left[\log p_\theta(x_i, z_i) - \log q_i(z_i)\right]}_{\equiv B_i(q_i,\theta)\ \text{local ELBO}}$$

we can rewrite the ELBO as a sum of **local ELBOs**.

Fixing the parameters, the ELBO is maximized and the bound is tight when:

$$q_i(z_i) = p_\theta(z_i \mid x_i)$$

Unlike in previous models we've looked at though, the exact posterior is intractable. (Question: why?)

Fixed-form VI

While the optimal update to q(zi) is not known, we can still constrain q(zi) to be Gaussian (or any other parametric family) and find its parameters that maximize the ELBO. Fixing the form of q(...) is called **fixed-form variational inference.**

$$q_i(z_i) = q_{\lambda_i}(z_i) = \mathcal{N}(z_i;\ \underbrace{\mu_i,\ \operatorname{diag}(\sigma_i^2)}_{\equiv \lambda_i\ \text{variational parameters}})$$

$$\underset{\lambda}{\operatorname{argmax}}\ B(q_\lambda, \theta)$$

Variational EM

Variational EM is a version of EM where the E-step is replaced with a variational approximation. As said before, we cannot form the exact posterior of z, conditional on parameters, so we cannot perform the exact E step. Instead we will replace it with updating the variational parameters of q(Z).

vEM: until convergence

M-step:

$$\theta \leftarrow \arg\max_{\theta} B(q_\lambda, \theta)$$

E-step:

$$\lambda \leftarrow \arg\max B(q_\lambda, \theta)$$

Unfortunately, neither of these steps have closed-form solutions, so we will have to use **gradient ascent.**

M-step with stochastic gradient ascent

$$\theta_t \leftarrow \theta_{t-1} + \epsilon_t \nabla_\theta B(q_\lambda, \theta)$$

Each iterate takes a small step along the gradient. The gradient can be written as the sum of gradients of the local ELBOs:

$$\nabla_\theta B(q_\lambda, \theta) = \sum_{i=1}^{n} \nabla_\theta B_i(q_{\lambda_i}, \theta)$$

When the data set is large, we can replace this with an unbiased estimate of the gradient by subsampling:

$$= \frac{n}{n} \sum_{i=1}^{n} \nabla_\theta B_i(q_{\lambda_i}, \theta) = n \mathbb{E}_i[\nabla_\theta B_i(q_{\lambda_i}, \theta)]$$

$$\approx n \nabla_\theta B_i(q_{\lambda_i}, \theta), \quad i \sim \text{Uniform}(1..n)$$

Each local ELBO is itself an expectation with respect to q:

$$= n \nabla_\theta \mathbb{E}_{z_i \sim q_{\lambda_i}} [\log p_\theta(x_i, z_i) - \log q_{\lambda_i}(z_i)]$$

The gradient pushes in and everything that is constant wrt theta drops:

$$= n \mathbb{E}_{z_i} [\nabla_\theta \log p_\theta(x_i, z_i)]$$

The expectation of the gradient is not tractable, again because the gradient is complex. However, just as we replaced the gradient above with an unbiased expectation, we can do the same here:

$$\approx \frac{n}{M} \sum_{m=1}^{M} \nabla_\theta p_\theta(x_i, z_i^{(m)}), \quad i \sim \text{Uniform}(1...n), \quad z_i^{(m)} \overset{iid}{\sim} q_{\lambda_i}(z_i)$$

As we saw last time, following these stochastic gradients (which are unbiased estimators for the exact gradient) will converge to a local mode, under the condition that the step sizes follow the Robbins-Monro conditions:

$$\sum_{t=1}^{\infty} \epsilon_t = \infty, \quad \sum_{t=1}^{\infty} \epsilon_t < \infty$$

Computing gradients automatically with the backpropagation algorithm

We still have to compute the gradient of the complete data log-likelihood

$$\nabla_\theta \log p_\theta(x_i, z_i)$$

where recall that the parameters are all the weights and biases of the multilayer perception:

$$\equiv \sum_{i=1}^{n} \nabla_{W,b} \log \mathcal{N}(x_i \mid g_L(\cdots g_2(a_1(W_1 z + b_1)) \cdots), I)$$

This is a complicated gradient that we won't solve by hand. However, as long as we can implement the complete data log-likelihood as a function in PyTorch (or another framework for automatic differentiation) then we can **evaluate the gradient automatically** using the backpropagation algorithm (backprop).

Backprop is not just the chain rule of calculus. It is a dynamic programming algorithm that cleverly evaluates the gradient in time complexity equal to the time complexity of the original function! Since our log-likelihood in this case is just a sum of Gaussian PDFs, we can evaluate it efficiently, and thus evaluate its gradient efficiently.

Variational E-step

As with the M-step, we will be doing gradient ascent for the E-step. First let's redefine the variational parameters so that they are unconstrained:

$$\lambda_i = (\mu_i, \log \sigma_i^2) \in \mathbb{R}^{2d}$$

Now we can think about taking gradients with respect to the local variational parameters:

$$\nabla_{\lambda_i} B(q_{\lambda_i}, \theta) = \nabla_{\lambda_i} \mathbb{E}_{z_i \sim q_{\lambda_i}} \left[\log p_\theta(x_i, z_i) - \log q_{\lambda_i}(z_i) \right]$$

Question: why doesn't this gradient push into the expectation?

Reparameterization gradients aka pathwise gradients

The expectation is with respect to the variational distribution, which is Gaussian. Notice that we can draw samples from a Gaussian with the following algorithm:

$$z \sim \mathcal{N}(\mu, \operatorname{diag}(\sigma^2)) \iff \begin{aligned} \epsilon &\sim \mathcal{N}(0, I) \\ z &= \mu + \sigma \epsilon \equiv R(\lambda, \epsilon) \end{aligned}$$

Notice on the RHS we have rewritten z in terms of its **reparameterization**: a deterministic function terms of parameters and noise, where the noise is drawn from a distribution that does not depend on the parameters.

The law of the unconscious statistician (LOTUS) generally says: $\mathbb{E}_{x \sim p}[f(x)] = \int f(x) p(x) \, dx$

Therefore we can rewrite the expectation above as:

$$\mathbb{E}_{z_i \sim q_{\lambda_i}}[h(z_i)] = \mathbb{E}_{\epsilon_i \sim \mathcal{N}(0, I)}[h(R(\lambda_i, \epsilon_i))], \quad h(z_i) = \log p_\theta(x_i, z_i) - \log q_{\lambda_i}(z_i)$$

Notice that now the expectation does not depend on the parameters that we're taking the gradient with respect to. So the gradient pushes in!

$$\nabla_{\lambda_i} \mathbb{E}_{\epsilon_i}\left[h(R(\lambda_i, \epsilon_i))\right] = \mathbb{E}_{\epsilon_i}\left[\nabla_{\lambda_i} h(R(\lambda_i, \epsilon_i))\right]$$

Once again, the expectation of this gradient is not available in closed-form. But we can approximate it:

$$\approx \frac{1}{M}\sum_{m=1}^{M} \nabla_{\lambda_i} h\left(R(\lambda_i, \epsilon_i^{(m)})\right), \quad \epsilon_i^{(m)} \overset{iid}{\sim} \mathcal{N}(0, I)$$

Now all we need to be able to do is evaluate the gradient; but we can again do this using backpropagation. That's the magic of reparameterization gradients!

Putting it all together:

M-step: for $t = 1 \dots T$

$i \sim \text{Uniform}(1 \dots n)$

$z_i^{(m)} \overset{iid}{\sim} q_{\lambda_i}(z_i), \quad m = 1 \dots M$

$$\theta \leftarrow \theta + \epsilon_t \frac{n}{M} \sum_{m=1}^{M} \nabla_\theta \log p_\theta(x_i, z_i)$$

E-step: for $t = 1 \dots T$

for $i = 1 \dots n$

$$\epsilon_i^{(m)} \overset{iid}{\sim} \mathcal{N}(0, I), \quad m = 1 \dots M$$

$$\lambda_i \leftarrow \lambda_i + \epsilon_t \frac{1}{M} \sum_{m=1}^{M} \nabla_{\lambda_i} h\left(R(\lambda_i, \epsilon_i^{(m)})\right)$$

ELBO:

$$\mathcal{B}(q_\lambda, \theta) = \sum_{i=1}^{n} \mathbb{E}_{z_i \sim q_{\lambda_i}}[h(z_i)]$$

$$\approx \sum_{i=1}^{n} \frac{1}{M} \sum_{m=1}^{M} h(R(\lambda_i, \epsilon_i^{(m)}))$$

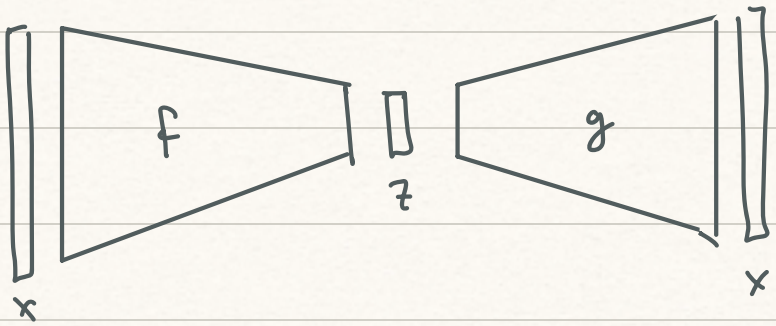
Amortized variational inference and VAEs

Notice that the E-step involves finding the variational parameters for every data point. When we have many data point, this computation adds up. Instead we can introduce a **recognition network**. This is another neural network that takes in a data point and spits out a prediction of its variational parameters:

$$f_\phi(x_i) \equiv f_L\left(f_{L-1}(\cdots f_1(x_i)\cdots)\right) \approx x_i$$

ϕ: weights and biases of f

Typically the architecture of the recognition network will be the mirror image of the **generative network**



Another term for these two networks are the **encoder** and **decoder** networks. These two networks together are called an **autoencoder** (because it maps x back to itself="auto"). Autoencoders have long history in machine learning and AI. In this setting we have a **variational autoencoder (VAE)**, because the first work parameterizes q(z | x) and the second parameterizes p(x | z). VAEs were concurrently invented by Kingma & Welling (2014) and Rezende et al. (2014).

This recognition network is shared across data points. So we can then replace the E-step above with one that subsamples the data and updates the weights and biases of the recognition network using stochastic gradients:

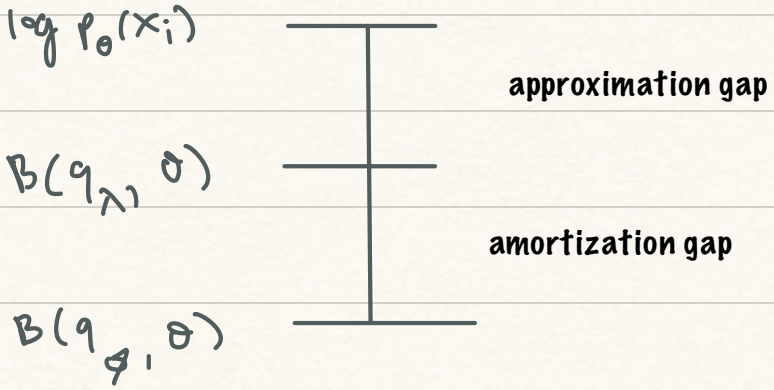
E-step: for $t = 1 \ldots T$

$i \sim \text{Uniform}(1 \ldots n)$

$\epsilon_i^{(m)} \overset{iid}{\sim} \mathcal{N}(0, I)$, $m = 1 \ldots M$

$$\phi \leftarrow \phi + \epsilon_t \frac{1}{M} \sum_{m=1}^{M} \nabla_\phi h\left(R\left(f_\phi(x_i), \epsilon_i^{(m)}\right)\right)$$

Amortization inference significantly improves computation but at the expense of approximation error. See the papers of Cremer et al. (2018) and Margossian and Blei (2024).



VAEs as information bottlenecks; connection to PCA

What's the point of a (variational) autoencoder which just maps data back to itself. One way to understand the value is as understanding a (V)AE as an **information bottleneck**. It takes in x, passes it thought a bottleneck, and then tries to reconstruct x. In so doing, we hope that it **learns meaningful structure.**

To see this, consider **principal components analysis (PCA)**, the most common tool for learning structure in data:

$$\mathbb{E}[x_i] = \hat{x}_i = W z_i$$

PCA assumes that the expectation of each data point (a vector) is equal to a linear function of a parameter matrix W and a parameter vector zi. PCA can be understood as learning W and Z by minimizing squared reconstruction loss subject to the condition that W is an orthogonal matrix:

$$\underset{W,z}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \| x_i - W z_i \|_2^2 \quad \text{s.t. } W \in O \quad = \quad \underset{W,z}{\operatorname{argmax}} \prod_{i=1}^{n} \mathcal{N}(x_i; W z_i, I) \quad \text{s.t. } W \in O$$

As the RHS shows, this is just a probabilistic matrix factorization algorithm with a Gaussian likelihood.

If we knew W, and a new data point x_i arrived, what would our estimate of z_i be?

$$\hat{x}_i = W z_i$$

$$W^T \hat{x}_i = W^T W z_i \quad \text{note that } W^T W = I$$

$$W^T \hat{x}_i = z_i \longrightarrow \underset{z_i}{\operatorname{argmin}} \| x_i - W z_i \|_2^2 = W^T x_i$$

We can therefore rewrite the PCA algorithm as simply learning an orthogonal matrix W such that:

$$\underset{W \in O}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \| x_i - W W^T x_i \|_2^2$$

This loss is small when:

$$x_i \approx W W^T x_i$$

So PCA is just a single linear-layer autoencoder!

More generally, a very large class of methods for unsupervised learning can be understood as imposing some kind of information bottleneck, such that the parameters are forced to represent common / meaningful structure in the data.

The "bits-back" argument

Let's say you want to send messages x_i and encode them such that the message length is minimized. We know from the source coding theorem that the best possible coding scheme would have message lengths of:

$$\ell(C^*(x_i)) = \log_2 \frac{1}{p(x_i)}$$

↑ optimal coding scheme

What if p(x) is unknown and intractable? For instance, what if x is an image from the set of all images? Let's instead represent it as a mixture distribution of tractable components:

$$p(x_i) = \sum_{z_i} p_\theta(x_i \mid z_i)\, p_\theta(z_i)$$

For a given z_i, we can code x_i according to p(x_i | z_i). The length of that code is:

$$\ell(C(x_i \mid z_i)) = \log_2 \frac{1}{p_\theta(x_i \mid z_i)}$$

Assume the sender and receiver both have access to P(x, z), so the receiver can decode x. The sender will have to transmit the z_i they used to encode x_i so that the receiver can decode x_i. Sending z_i will cost extra bits. How many?

$$\log \frac{1}{p_\theta(z_i)} + \log \frac{1}{p_\theta(x_i \mid z_i)}$$

Which z_i should the sender use to encode x_i? One possibility would be:

$$z_i = \underset{z_i}{\operatorname{argmax}}\ p_\theta(x_i \mid z_i)$$

This would minimize the number of bits used to encode x_i | z_i. However it might then require many bits to encode z_i, if the selected value has low probability under p(z_i).

Instead, what if the sender **sampled** z_i from the distribution p(z_i | x_i). (Assume for now they can.)

$$z_i \sim p_\theta(z_i \mid x_i)$$

Sampling from a distribution involves using **random bits**. This is a resource. The key idea with "bits-back" is that the receiver can **recover the random bits** that the sender used to sample z_i.

Sender:
1. $z_i \sim p_\theta(z_i \mid x_i)$
2. encode z_i using $p_\theta(z_i)$, x_i using $p_\theta(x_i \mid z_i)$
3. send $C(z_i, x_i)$

Receiver:
1. decode z_i using $p_\theta(z_i)$, x_i using $p_\theta(x_i \mid z_i)$
2. recover random bits using $p_\theta(z_i \mid x_i)$

So the total cost of the message, accounting for the bits back cost is:

$$\log_2 1/p_\theta(z_i) + \log_2 1/p_\theta(x_i \mid z_i) - \log_2 1/p_\theta(z_i \mid x_i) = \log_2 1/p_\theta(x_i)$$

↑ cost to encode z_i ↑ cost to encode x_i | z_i ↑ bits back ↑ optimal rate!

So if the receiver recovers the random bits, and then uses them to encode their message back, then we achieve the optimal rate of compression. This is the "bits-back argument", which has been used to motivate latent variable models.

One problem with this argument is that if P(x_i) is intractable then so is P(z_i | x_i). So we can't actually do what we just said. This is where we can introduce an approximate **decoder**:

$$q_\phi(z_i \mid x_i) \approx p_\theta(z_i \mid x_i)$$

We will want to fit the parameters of this decoder so that it is close to exact (intractable) posterior. We will still encode messages according to the joint p(Z_i, X_i); we will therefore call this the **encoder**.

What's the total cost of a message, using the decoder network instead of the exact posterior?

$$\log_2 1/p_\theta(z_i) + \log_2 1/p_\theta(x_i \mid z_i) - \log_2 1/q_\phi(z_i \mid x_i)$$

↑ cost to encode z_i ↑ cost to encode x_i | z_i ↑ bits back

If the sender uses q(z_i | x_i) to sample z_i then we can also think about the **expected** cost of a message:

$$-\mathop{\mathbb{E}}_{z_i \sim q_\phi(z_i)}\left[\log \frac{p_\theta(z_i, x_i)}{q_\phi(z_i)}\right] = -\text{ELBO}(q_\phi, p_\theta)$$

So if we wanted to obtain a coding scheme (i.e., encoder and decoder) that minimizes the expected length of messages, we would just be fitting a variational autoencoder! The moral of this story is that compression and probabilistic modeling are the same thing, in a certain sense.