Variational Autoencoders

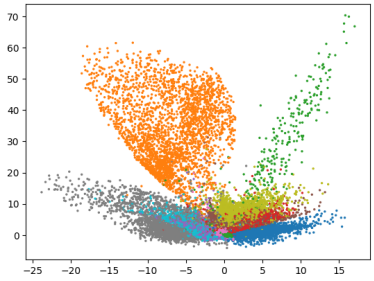
Muchang Bahng

Spring 2023

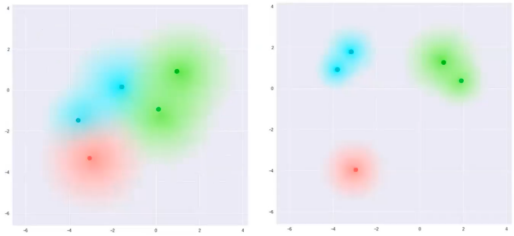
Contents

1 Variational Autoencoders

Note that like linear latent variable models such as PCA, autoencoders "encode" our samples in a latent space, which we will call $\mathcal{Z}$. If we wanted to create a generative model from autoencoders, we can use the analogous transition from PCA to PPCA, by changing our points to distributions. Like in PPCA, we might want to define a standard Gaussian over the latent space $\mathcal{Z}$ and transform this into the original space $\mathcal{X}$. However, there is a small problem with autoencoders. The latent space where the encoded vectors lie may not be contiguous, which means that the distribution of the latent space may not be very simple either. Look at the encodings of MNIST below. Trying to sample from this space with a isotropic Gaussian results in a high probability of hitting the "dead" zones which may give garbage generative results. If the space has discontinuities and you sample a variation from there, the decoder will simply generate an unrealistic output.



(a) Training an autoencoder on MNIST and then visualizing the encodings from a 2D latent space shows the formation of distinct clusters, but there are huge empty spaces where the labeling may be ambiguous and not allow us to interpolate effectively.



(b) Therefore, we want the encodings to be contiguous while still being distinct. This allows smooth interpolation which is easy to sample from with a simple pdf, e.g. Gaussian, and enables construction of *new* samples.

Figure 1

In 2013, Kigma introduced this generative model that bridges between latent variable models and deep neural nets. We can think of the relationship between the linear model PCA and its latent counterpart PPCA as the relationship between the nonlinear autoencoders and variational autoencoders. This is a good time to review linear and nonlinear latent variable models in my ML notes. Note that in a LVM, the family of functions $\{D_\beta\}$ is used to map a latent variable z to the likelihood $p(x \mid z)$. This can be done by a direct transformation of the random variable Z (e.g. PPCA or as we will see later, *normalizing flows*), or we can have $D_\beta(z)$ be an explicit parameterization (e.g. Gaussian mixture models and variational autoencoders). Given neural networks, we can do the latter method quite easily, and we are already familiar with the architectures to do so.

Example 1.1 (Classification Nets Parameterize Multinomials)

In a classification neural network with parameters β, it takes in an input x and outputs a softmax vector $f_\beta(x) = (p_1, \dots, p_K)^T$. This basically means that $f_\beta(x) = \theta$ parameterizes the conditional distribution (in this case, multinomial) of y given x.

$$Y \mid X = x \sim \text{Multinomial}(\theta = f_\beta(x)) \tag{1}$$

This is a much more efficient way to store conditional distributions than a $\dim(X)(K-1)$ lookup table.

Therefore, by generating a latent variable $p(z)$ (that is simple and fixed), we can use a deep neural network D_β to generate $\theta = D_\beta(z)$ which serve as the parameters of the conditional distribution $p_\theta(x \mid z)$, and then sampling from this distribution is easy because we assume that $p_\theta(x \mid z)$ is in an explicitly parameterized

family of distributions. This allows us to easily sample from the joint distribution.

$$p_\theta(x, z) = p_\theta(x \mid z)\, p(z) \tag{2}$$

This is all great, but computing

$$p(x) = \int p(x, z)\, dz = \int p_\theta(x \mid z)\, p(z)\, dz \tag{3}$$

is computationally intractable. With strong assumptions, like conditional independence of not just $p(x \mid z)$ but *also* $p(z \mid x)$, in RBMs, we can construct pretty good approximations. Recall that for RBMs, the derivative of the log of $p(x)$ decomposes into a positive phase that requires you to integrate over $p(z \mid x)$, which is easy, and $p(x, h)$, which is hard to do in general. But through contrastive divergence, we can approximate the integral by sampling a $\tilde{x}$ and constructing another integral over $p(z \mid x)$, which is then easy to compute.

Example 1.2 (Bernoulli-Bernoulli RBM)

We would like to approximate a d-dimensional Bernoulli vector x with a latent variable $z \in \mathbb{R}^K$. We will assume a prior $p(z) \sim N(0, I)$, and let us have a neural net D_β that parameterizes the random vector x, where $x_i \sim \text{Bernoulli}(\theta_i)$ for θ_i. Then, by conditional independence,

$$p(x \mid z) = \prod_{i=1}^{d} p_\theta(x_i \mid z) = \prod_{i=1}^{d} \theta_i^{x_i}(1-\theta_i)^{1-x_i} = \prod_{i=1}^{d} \big[D_\beta(z)\big]_i^{x_i}\big(1 - [D_\beta(z)]_d\big)^{1-x_i} \tag{4}$$

and we can see that since p has the flexibility of whatever vector in $[0,1]^D$ it can be captured by the neural net D_β. It encompasses a broad family of Bernoulli probability distributions. We can see that we have some method to compute $p(x \mid z)$. We train a neural net (somehow) and do forward prop on it to generate the correct parameters modeling the distribution of x. Note that in RBMs the conditional independence allowed us to integrate over z easily. To see why, in the example above, the integral becomes

$$p(x) = \sum_{z \in \{0,1\}^k} \left\{ \prod_{i=1}^{d} \big[D_\beta(z)\big]_i^{x_i}\big(1 - [D_\beta(z)]_d\big)^{1-x_i} \right\} p(z) \tag{5}$$

where k is the dimension of the latent space. However, integrating over all z's for more complex spaces is not feasible.

D_β is clearly nonlinear and we can't just do some simple closed form optimization, so we must use the tools introduced in nonlinear latent variable models. Namely, we revisit the variational lower bound. Recall that estimating the true $p(x)$ can be reduced to the problem of finding a good approximate of the true $p(z \mid x)$, the *posterior* or *inference component*, with some family of distributions $\{q_\phi\}$. Just like the generation model, we can build another neural network E_α such that $\phi = E_\alpha(x)$ parameterizes the conditional distribution of z, called our *encoder*.

$$p(z \mid x) \approx q_\phi(z \mid x) = q_{E_\alpha(x)}(z) \tag{6}$$

Example 1.3 (Encoder Neural Net Generates Parameters of Likelihood)

If $\phi = (\mu, \sigma)$, where σ is just the vector representing variances of independent Gaussians, then we can use the neural network E to get $\phi = E_\alpha(x)$. In the example, $\phi = (\mu, \log \sigma^2)$ since we want to allow negative values, and

$$q_\phi(z \mid x) \sim \mathcal{N}(E_\alpha(x)) = \mathcal{N}(\phi) = \mathcal{N}(\mu, \sigma^2) \tag{7}$$

Therefore, by modeling both the likelihood and posterior with probability distributions that can be parameterized by an output of neural networks, we have a *variational autoencoder*.

Definition 1.1 (Variational Autoencoders)

In a **variational autoencoder (VAE)**, we assume that the covariates $x^{(i)} \sim X$ are generated as the marginal of a joint distribution $p(x, z)$ with latent variable $z \sim X$. We model the sub-distributions as such:

1. We assume that the prior $p(z)$ is of fixed simple form, e.g. a standard Gaussian.
2. We assume that the likelihood $p(x \mid z)$ can be approximated by a parameteric family of distributions $p_\theta(x \mid z)$, where $\theta = D_\beta(z)$ is generated by a **decoder** neural network parameterized by β.
3. We assume that the posterior $p(z \mid x)$ can be approximated by a parameteric family of distributions $q_\phi(z \mid x)$,[a] where $\phi = E_\alpha(x)$ is generated by an **encoder** neural network parameterized by α.

We can optimize θ, λ by equivalently optimizing the parameters α, β of the nets. Once this is done,

1. To conduct inference, we can input in some data x and retrieve the distribution of its latent representation as $q_{E_\alpha(x)} \approx p(z \mid x)$.
2. To generate data, we can sample z from $p(z)$ and sample from $p_{D_\beta(z)}(x) \approx p(x \mid z)$.

[a] Note that this is an assumption! The posterior may be an arbitrary shape or form.

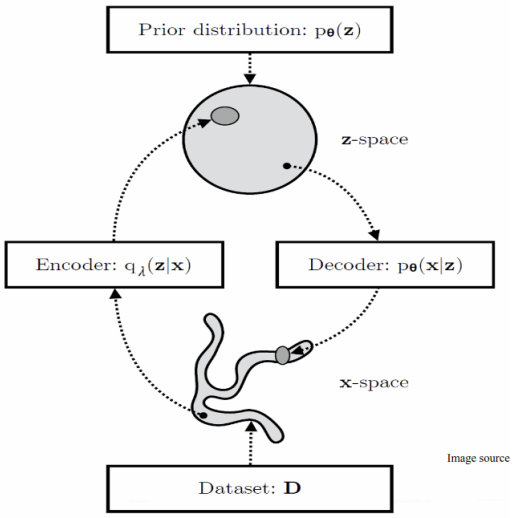


Figure 2: A pictorial diagram of a nonlinear latent variable model to refresh your memory.

So how do we actually train this? Just like with every other nonlinear latent variable models, we can use the evidence lower bound. We do a quick review. Recall that the KL divergence can be decomposed into the sum of conditional entropies, and hence we can use the fact the KL divergence is always nonnegative to create this bound.

$$\log p_\theta(x) = KL\big(q_\phi(z \mid x) \,||\, p_\theta(z \mid x)\big) + \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x, z)] - \mathbb{E}_{q_\phi(z|x)}[\log q_\phi(z \mid x)] \tag{8}$$

$$\geq \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x, z)] - \mathbb{E}_{q_\phi(z|x)}[\log q_\phi(z \mid x)] = \text{ELBO}(x, \phi, \theta) \tag{9}$$

and therefore by summing over all the data points we can get the evidence lower bound, which holds for any set of distributions $q_\phi^{(1)}, \dots, q_\phi^{(N)}$.

$$\sum_{i=1}^{N} \log p_\theta(x^{(i)}) \geq \sum_{i=1}^{N} \mathbb{E}_{q_\phi(z|x^{(i)})}[\log p_\theta(x^{(i)}, z)] - \sum_{i=1}^{N} \mathbb{E}_{q_\phi(z|x^{(i)})}[\log q_\phi(z \mid x^{(i)})] = \text{ELBO}(\mathcal{D}, \phi, \theta) \tag{10}$$

Therefore, minimizing the KL divergence is equivalent to maximizing the ELBO w.r.t. θ and ϕ. As we have seen in my ML notes, the gradient of the ELBO w.r.t. θ be solved by computing the gradient directly and using SGD. However, taking the gradient w.r.t. ϕ is more complicated since we cannot put the gradient in the expectation (since we are deriving and integrating w.r.t. ϕ). We can use the log-derivative trick[1], but this is known to have high variance and is not sufficient to backpropagate huge neural nets. This is not actually the biggest problem either, as we will see that the extra parameters we've introduce with the neural nets do not allow us to backpropagate across stochastic variables.

Now we talk about more implementation details presented in the paper, which is a simplified form.

1. The first is that [KW22] fixes a prior isotropic multivariate Gaussian $p_\theta(z) = \mathcal{N}(0, I)$, which lacks parameters.
2. It also assumes that $p_\theta(x \mid z)$ be a multivariate Gaussian (for real-valued data) or Bernoulli (in binary data). This gives us a nice parameteric form that may hopefully approximate the true posterior, which may be extremely complicated.

$$p_\theta(x \mid z) = \mathcal{N}(\mu^{(i)}, \sigma^{2(i)} I) \qquad \theta = \{\mu^{(i)}, \sigma^{2(i)}\} = D_\beta(z^{(i)}) \tag{11}$$

$$p_\theta(x \mid z) = \text{Bernoulli}(\theta) \qquad \theta = D_\beta(z^{(i)}) \tag{12}$$

3. It finally assumes that the true posterior $p_\theta(z \mid x)$, which is intractable, takes on a form $q_\phi(z \mid x)$ that can be approximated by a Gaussian with diagonal covariance.

$$q_\phi(z \mid x^{(i)}) = \mathcal{N}(\mu^{(i)}, \sigma^{2(i)} I) \qquad \phi = \{\mu^{(i)}, \sigma^{2(i)}\} = E_\alpha(x^{(i)}) \tag{13}$$

So let's walk through the forward propagation with the likelihood set to Bernoulli. We have a dataset $\{x^{(i)}\}$. For each $x^{(i)}$

1. We put it into our encoder network and get out the parameters $\phi^{(i)} = E_\alpha(x^{(i)})$ for our latent Gaussian. We output the mean $\mu_\alpha^{(i)}$ and either the variance $(\sigma_\alpha^{(i)})^2$, or the log-variance $\log(\sigma_\alpha^{(i)})^2$.[2]
2. We sample a $z^{(i)}$ from this Gaussian and feed it into the decoder network $D_\beta(z^{(i)})$, which returns the parameters $\theta^{(i)} = D_\beta(z^{(i)})$ for our Bernoulli likelihood.
3. We sample $\hat{x}^{(i)}$ from this Bernoulli and retrieve another sampled version of our input.

Great, so we can draw a computation graph for this, but before we do, let's revisit the basics and draw a simpler example.

Example 1.4 (Softmax)

In the graph below, note that our "prediction" $\hat{y}^{(i)}$ for the data point $x^{(i)}$ is not really a prediction at all. If we have three categories (cat, dog, fish) and I give you an image, the model will output a vector of probabilities, e.g. $\hat{y}^{(i)} = [0.1, 0.2, 0.7]$. While we can easily infer that the probability of a fish is highest, and therefore should be our prediction, the model didn't actually output one answer. Rather, we should interpret this output vector $\hat{y}^{(i)}$ as a *parameter* θ of a multinomial distribution. From this we can *sample* an actual prediction $\hat{y}^{(i)} \sim \text{Multinomial}(\theta)$.
When we compute the loss of a sample, we compute the likelihood by taking the true output $y^{(i)}$ and one-hot encoding it. One-hot encoding allows us to convert the label $y^{(i)}$ into a parameter of a

[1] explained in my ML notes

[2] Since variances must always be positive, sometimes a slight tweaking of transformations is needed. For example, if an encoder network has a final linear layer, then the variances may not be always positive, which is a problem, so we may do an exponential or a ReLU at the end. You may notice that the log may often return negative numbers, but as we will see later, we can add a corresponding transformation in our *reparamaterization trick*, which will exponentiate the log-variances (this is how it's implemented in the code) before using them as variances. Calculating the log-variance first and then exponentiating it may also bring better numerical stability.

multinomial itself, and this process is hidden inside ℓ. Therefore we want to maximize

$$\log p(y^{(i)} \mid x^{(i)}) = p_{\theta^{\hat{(i)}}}(y^{(i)}) = \sum_j y_j^{(i)} \log \hat{\theta}_j^{(i)} \tag{14}$$

which is the cross entropy.

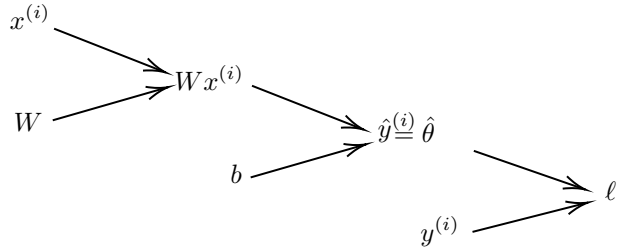


Figure 3: When we draw a computation graph to visualize the backpropagation of a softmax regression function, note that we need to have the gradients start from the log-likelihood ℓ and reach "back" to W and b, our model parameters.

Another way to approximate the log likelihood ℓ is to simply take the output parameter $\hat{\theta}^{(i)}$, sample y's from it K times, penalizing when needed, and averaging the penalties.[a]

$$\log p(y^{(i)} \mid x^{(i)}) = \log \mathbb{E}_{y \sim \hat{\theta}^{(i)}} \left[\mathbb{1}_{y^i = y} \right] \tag{15}$$

$$= \log \int \mathbb{1}_{y^{(i)}=y} p_{\hat{\theta}^{(i)}}(y)\, dy \tag{16}$$

$$\approx \log \left\{ \frac{1}{K} \sum_{k=1}^{K} \mathbb{1}_{y^{(i)}=y^{(i,k)}} \right\} \tag{17}$$

where $y^{(i,k)} \sim \text{Multinomial}(\hat{\theta}^{(i)})$.

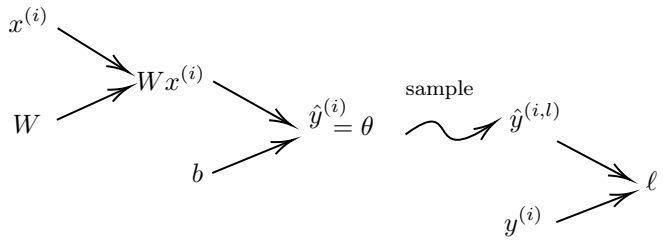


Figure 4: Note that since it's much harder to sample and converge onto a good approximation, it's tougher to go through the curly arrow. That is, it is harder to backpropagate through **stochastic nodes**.

Therefore, we can think of backpropagating as either one of two things. Either backpropagate straight through the parameter like we did first, or sample from the estimated distribution and sum up to approximate the samples.

[a] This is useful if we can easily sample, but do not know the closed form of the likelihood distribution.

Let's keep this in the back of our mind, and focus on the computation for a bit. By the simplifying form that q_ϕ is a Gaussian, we can even get a nearly closed form of the ELBO.

Theorem 1.1 ()

Given the assumptions above,

$$\text{ELBO}(x, \phi, \theta) \coloneqq \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x \mid z)] + \mathbb{E}_{q_\phi(z|x)}[\log p(z)] - \mathbb{E}_{q_\phi(z|x)}[\log q_\phi(z \mid x)] \tag{18}$$

$$= \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x \mid z)] + \frac{1}{2}\sum_{d=1}^{D}\left(1 + \log((\sigma_d)^2) - (\mu_d)^2 - (\sigma_d)^2\right) \tag{19}$$

Therefore, the derivative of last two terms are *exact*, i.e. we don't need to estimate them with SGD.

Proof. Let's evaluate the last two expectations.

$$\mathbb{E}_{q_\phi(z|x)}[\log p(z)] = \int \log p(z) \cdot q_\phi(z \mid x)\, dx \tag{20}$$

$$= \int \log\{\mathcal{N}(z \mid 0, I)\} \cdot \mathcal{N}(z \mid \mu, \sigma^2)\, dz \tag{21}$$

$$= -\frac{D}{2}\log(2\pi) - \frac{1}{2}\sum_{d=1}^{D}\mu_d^2 + \sigma_d^2) \tag{22}$$

$$\mathbb{E}_{q_\phi(z|x)}[\log q_\phi(z \mid x)] = \int \log q_\phi(z \mid x)\, q_\phi(z \mid x)\, dz \tag{23}$$

$$= \int \log\{\mathcal{N}(z \mid \mu, \sigma^2)\}\mathcal{N}(z \mid \mu, \sigma^2)\, dx \tag{24}$$

$$= -\frac{D}{2}\sum_{d=1}^{D} 1 + \log(\sigma_d^2) \tag{25}$$

Corollary 1.1 (Bernoulli Likelihood)

If $p\theta(x \mid z)$ is Bernoulli, then the ELBO can be estimated by estimating the likelihood

$$\text{ELBO}(x, \phi, \theta) = \frac{1}{L}\sum_{l=1}^{L} + \frac{1}{2}\sum_{d=1}^{D}\left(1 + \log((\sigma_d)^2) - (\mu_d)^2 - (\sigma_d)^2\right) \tag{26}$$

With these two things, let's take a look at the computation graph of the VAE.

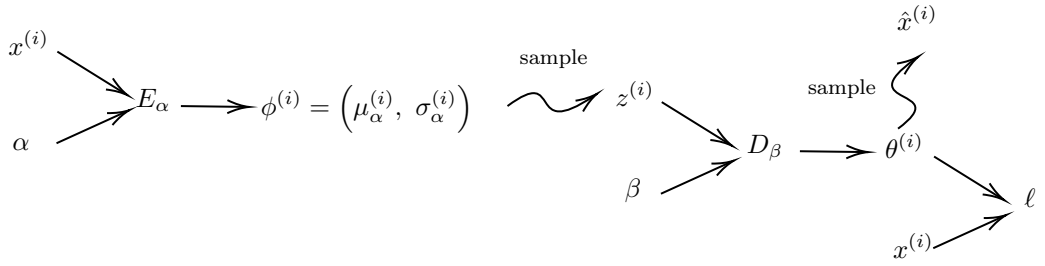


Figure 5: Computational graph of a VAE.

Note that since we have shown that the derivative of the last two expectations (i.e. the derivative of the KL divergence) is dependent on the outputs σ and μ of the neural net, we must also backpropagate amongst them as well, which is why you see the giant arrow from ϕ to ℓ.

We can see that since we know the closed form of $p_\theta(x \mid z)$, backpropagating over $\theta^{(i)}$ is no problem. However, we must reach α to update the neural network weights, and to do this we must go backprop through the stochastic node $z^{(i)}$, which may stump you. In fact, how do you even backpropagate against something that's a random variable? Remember that in our softmax example, we could simply compute the log likelihood of a bunch of samples at this node to get an approximate average log likelihood. This is a start, but in this case we want to compute the *gradient* of the log likelihood. Fortunately, the log-derivative trick allows us to swap the gradient and expectation under a certain form, putting the expectation on the outside and therefore allowing us to sample gradients.

$$\nabla_\phi \mathbb{E}_{q_\phi(z)}[f(z)] = \mathbb{E}_{q_\phi(z)}[f(z) \nabla_\phi \log q_\phi(z)] \tag{27}$$

So we apply the same idea by sampling $z^{(1)}, \dots, z^{(L)}$ from the Gaussian with parameters $\phi^{(i)}$ and for each sample we fix $z^{(l)}$, allowing us to backpropagate to α to get a noisy estimate. By averaging these noisy gradients we have an approximation of the true gradient w.r.t. $x^{(i)}$.[3] Unfortunately, when doing this for gradients the variance tends to be very high up to a point where it is impractical, and even with a lot of sampling this frequently undershoots or overshoots the actual gradient. Therefore, we must come up with an alternative solution to estimate the gradient efficiently.

1.1 Reparamaterization Trick

In 2013, Durk Kigma, then a PhD student in machine learning at the University of Amsterdam, introduced a solution in [KW22] called the *reparamaterization trick*, which is just an application of a change of basis.

Theorem 1.2 (Reparameterization Trick)

Let $q_\phi(z|x)$ be a continuous distribution that can be sampled by first sampling $\epsilon \sim p(\epsilon)$ from a parameter-free distribution and then applying a differentiable transformation $g_\phi(\epsilon, x)$. Then:

$$\nabla_\phi \mathbb{E}_{q_\phi(z|x)}[f(z)] = \nabla_\phi \mathbb{E}_{p(\epsilon)}[f(g_\phi(\epsilon, x))] = \mathbb{E}_{p(\epsilon)}[\nabla_\phi f(g_\phi(\epsilon, x))] \tag{28}$$

or in integrals,

$$\nabla_\phi \int f(z)\, q_\phi(z \mid x)\, dz = \nabla_\phi \int f(g_\phi(\epsilon, x))\, p(\epsilon)\, d\epsilon = \int \nabla_\phi f(g_\phi(\epsilon, x))\, p(\epsilon)\, d\epsilon \tag{29}$$

where f is any differentiable function.

Proof. We clearly state our assumptions:

- We have a distribution $q_\phi(z|x)$ that can be reparameterized using $g_\phi(\epsilon, x)$
- ϵ is sampled from a parameter-free distribution $p(\epsilon)$
- g_ϕ is differentiable with respect to ϕ
- f is any differentiable function

The first equality to prove is:

$$\nabla_\phi \mathbb{E}_{q_\phi(z|x)}[f(z)] = \nabla_\phi \mathbb{E}_{p(\epsilon)}[f(g_\phi(\epsilon, x))]$$

This follows directly from the reparameterization property. Since $z = g_\phi(\epsilon, x)$ where $\epsilon \sim p(\epsilon)$, we can rewrite any expectation over $q_\phi(z|x)$ as an expectation over $p(\epsilon)$:

$$\mathbb{E}_{q_\phi(z|x)}[f(z)] = \mathbb{E}_{p(\epsilon)}[f(g_\phi(\epsilon, x))]$$

Taking ∇_ϕ of both sides gives us the first equality. The second equality to prove is:

$$\nabla_\phi \mathbb{E}_{p(\epsilon)}[f(g_\phi(\epsilon, x))] = \mathbb{E}_{p(\epsilon)}[\nabla_\phi f(g_\phi(\epsilon, x))] \iff \nabla_\phi \int f(g_\phi(\epsilon, x)) p(\epsilon) d\epsilon = \int \nabla_\phi f(g_\phi(\epsilon, x)) p(\epsilon) d\epsilon \tag{30}$$

[3] This like a SGD inside a SGD, since we're sampling over minibatches of the samples $x^{(i)}$ as well.

To justify exchanging the gradient and integral, we invoke the Leibniz integral rule. This exchange is valid when $f \circ g_\phi$ is differentiable with respect to ϕ (given in our assumptions), the domain of integration (support of $p(\epsilon)$) is independent of ϕ, and there are suitable integrability conditions (which we assume hold). Then we exchange the gradient and expectation:

$$\nabla_\phi \int f(g_\phi(\epsilon, x))p(\epsilon)d\epsilon = \int \nabla_\phi f(g_\phi(\epsilon, x))p(\epsilon)d\epsilon \tag{31}$$

A crucial observation is that $p(\epsilon)$ does not depend on ϕ (it is parameter-free), which is essential for this exchange to work.

$$\nabla_\phi \mathbb{E}_{q_\phi(z|x)}[f(z)] = \nabla_\phi \mathbb{E}_{p(\epsilon)}[f(g_\phi(\epsilon, x))] = \mathbb{E}_{p(\epsilon)}[\nabla_\phi f(g_\phi(\epsilon, x))] \tag{32}$$

This result is fundamental to variational inference and deep learning, particularly in training Variational Autoencoders (VAEs). The key insight is that by reparameterizing the random variable z in terms of a parameter-free random variable ϵ, we can separate the stochastic sampling (which is not differentiable) from the deterministic transformation g_ϕ (which is differentiable).

The biggest advantage to this is that we although we do not avoid sampling to approximate the log likelihood, the rate at which it converges is much faster than simple sampling or the log-derivative trick, which is demonstrated in the example below.

Example 1.5 (Gradient of Expection of $f(x) = x^2$ w.r.t. Gaussian)

Assume we have a normal distribution q that is parameterized by ϕ, specifically $q_\phi(x) = N(\phi, 1)$. We want to solve the below problem

$$\min_\phi \mathbb{E}_q[x^2] \tag{33}$$

This is of course a rather silly problem and the optimal $\phi = 0$ is obvious. One way to calculate $\nabla_\phi \mathbb{E}[x^2]$ is using the log-derivative trick as follows

$$\begin{aligned}
\nabla_\phi \mathbb{E}_q[x^2] &= \nabla_\phi \int q_\phi(x)x^2 dx && (34)\\
&= \int x^2 \nabla_\phi q_\phi(x) \frac{q_\phi(x)}{q_\phi(x)} dx && (35)\\
&= \int q_\phi(x) \nabla_\phi \log q_\phi(x) x^2 dx && (36)\\
&= \mathbb{E}_q[x^2 \nabla_\phi \log q_\phi(x)] && (37)
\end{aligned}$$

For our example where $q_\phi(x) = N(\phi, 1)$, this method gives

$$\nabla_\phi \mathbb{E}[x^2] = \mathbb{E}_q[x^2(x - \phi)] \tag{38}$$

and we can sample from $q \sim N(\phi, 1)$. Reparameterization trick is a way to rewrite the expectation so that the distribution with respect to which we take the gradient is independent of parameter ϕ. To achieve this, we need to make the stochastic element in q independent of ϕ. Hence, we write x as

$$x = \phi + \epsilon, \quad \epsilon \sim N(0, 1) \tag{39}$$

Then, we can write

$$\mathbb{E}_q[x^2] = \mathbb{E}_\epsilon[(\phi + \epsilon)^2] \tag{40}$$

where $p(\epsilon)$ is the distribution of ϵ, i.e., $N(0, 1)$. Now we can write the derivative of $\mathbb{E}_q[x^2]$ as follows

$$\nabla_\phi \mathbb{E}[x^2] = \nabla_\phi \mathbb{E}_p[(\phi + \epsilon)^2] = \mathbb{E}_p[2(\phi + \epsilon)] \tag{41}$$

If we actually plot the variances, for sample sizes of $N = 10, 100, 1000, 10000, 100000$, we can see that the reparamaterization trick produces estimates where the variances of the sampling distribution is an order of magnitude smaller.

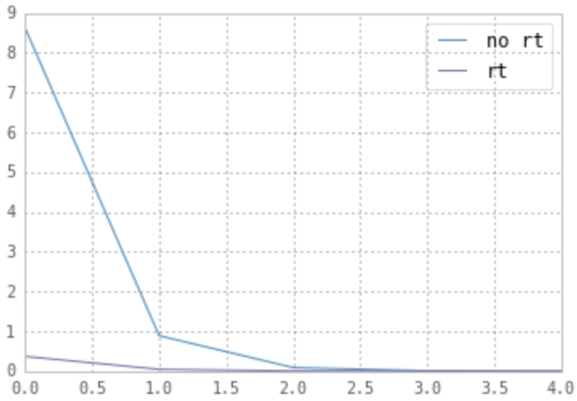


Figure 6: Credits to here.

Therefore, given that $\phi_\mu = \mu^{(i)}$ and $\phi_\sigma = (\sigma^{(i)})^2$ are the outputs of our encoder neural net, our transformation is

$$g_\phi(x, \epsilon^{(l)}) = \phi_\mu + \sqrt{\phi_\sigma} \odot \epsilon^{(l)} = \mu^{(i)} + \sigma^{(i)} \odot \epsilon^{(l)} \qquad \epsilon^{(l)} \sim \mathcal{N}(0, I) \tag{42}$$

or in the log-variance case, where the encoder outputs $\phi_\sigma = \log(\sigma^{(i)})^2 = 2 \log \sigma^{(i)}$, we have

$$g_\phi(x, \epsilon^{(l)}) = \mu^{(i)} + e^{\phi_\sigma / 2} \odot \epsilon^{(l)} = \mu^{(i)} + \sigma^{(i)} \odot \epsilon^{(l)} \qquad \epsilon^{(l)} \sim \mathcal{N}(0, I) \tag{43}$$

where $\odot$ is element-wise multiplication, can be used to model $p(z)$.[4] The intuition behind this is to "push" the stochastic node $z^{(i)}$ back so that we don't have to go through it to reach α. Therefore, to compute the noisy gradient for α, we can sample $\epsilon^{(l)} \sim \mathcal{N}(0, I)$ and treat it as a fixed variable when doing backpropagation, which essentially means that the mapping g_ϕ is a deterministic function (and so $g_\phi \circ E_\alpha$ is a deterministic neural net). Then we do this L times to get L gradients, and average them to get the unbiased estimate of the gradient of the single sample $x^{(i)}$. Then we must do this for all $x^{(i)}$ in a minibatch in SGD. Since this is still too computationally heavy, we usually set $L = 1$.

[4] This is called **factorized Gaussian posterior**, but it can be extended to the **full-variance Gaussian posterior** where the variance is parameterized through a lower triangular matrix L, from the Cholesky decomposition.

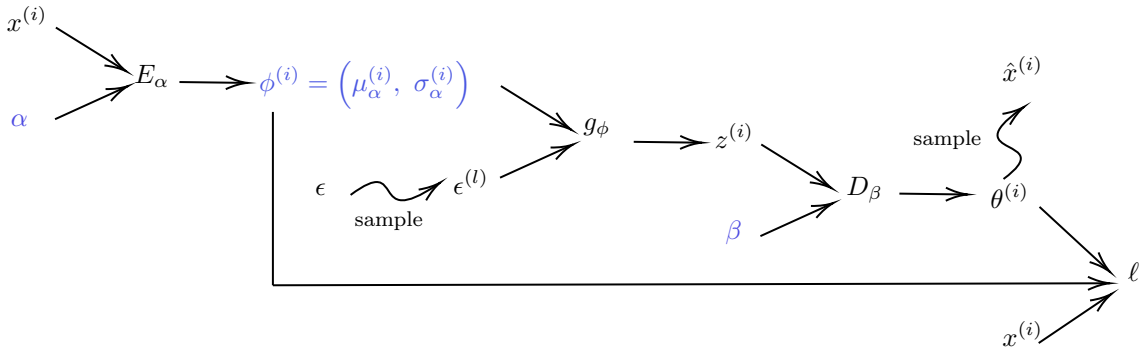


Figure 7: Note that the ϵ variable was added into the graph, and the blue variables are the parameters we want to backpropagate and optimize. Therefore the path from ℓ to α does not directly go through a stochastic node. Note that autograd calculates gradients for ϕ while on its way to α. But since ℓ is directly dependent on $\phi^{(i)}$, we want to make sure to update the $\phi^{(i)}$ directly with a step in the right direction. The rest of the parameters, while they do have their gradients computed, do not need to be updated with a step. A similar figure on a smaller scale can be found in Kigma's workshop presentation in NIPS.

Another advantage is that this is more general. Although we must meet the conditions that the posterior $p_\phi(z \mid x)$ should be an absolutely continuous distribution that can be modeled as the transformation of a simple random variable ϵ, since any distribution that is differentiable with respect to its parameters can be reparamaterized by inverting the multivariate CDF function and applying the implicit method.

In order to see how the gradients are calculated, recall the following theorem.

Theorem 1.3 (Change of Basis)

We have

$$\log q_\phi(z \mid x) = \log p(\epsilon) - \log \left| \det \left(\frac{\partial z}{\partial \epsilon} \right) \right| \tag{44}$$

where $\partial z / \partial \epsilon$ denotes the Jacobian matrix computed through $g_\phi(x, \epsilon)$.

Corollary 1.2 ()

This applied to the affine function g_ϕ above defined for the log-variance gives

$$\frac{\partial z}{\partial \epsilon} = \text{diag}(\sigma) \implies \log \left| \det \left(\frac{\partial z}{\partial \epsilon} \right) \right| = \sum_i \log \sigma_i \tag{45}$$

Corollary 1.3 (Auto-Encoding VB (AEVB) Algorithm)

Applying the reparamaterization trick to the gradient of the ELBO gives the following corollary.

$$\nabla_\phi \text{ELBO}(x, \phi, \theta) = \mathbb{E}_{p(\epsilon)}[\nabla_\phi \log p_\theta(x \mid z) - \nabla_\phi \log q_\phi(z \mid x) + \nabla_\phi \log p(z)] \tag{46}$$

Therefore we can unbiasedly approximate the gradient by sampling L points $\epsilon^{(1)}, \dots, \epsilon^{(L)}$ from $p(\epsilon)$, transforming them to $z^{(1)}, \dots, z^{(L)}$ through the function g_ϕ, and finally computing the gradient of the log posterior (which is easy since we know the closed form of the conditional distribution given z), and

finally averaging them.

$$\begin{aligned}\nabla_\phi \text{ELBO}(x, \phi, \theta) &\approx \frac{1}{L}\sum_{l=1}^{L}[\nabla_\phi \log p_\theta(x \mid z^{(l)}) - \nabla_\phi \log q_\phi(z^{(l)} \mid x) + \nabla_\phi \log p(z^{(l)})] \\ &= \frac{1}{L}\sum_{l=1}^{L}[\nabla_\phi \log p_\theta(x \mid g_\phi(\epsilon^{(l)}, x)) - \nabla_\phi \log q_\phi(g_\phi(\epsilon^{(l)}, x) \mid x) + \nabla_\phi \log p(g_\phi(\epsilon^{(l)}, x))]\end{aligned}$$

which is guaranteed to converge by the law of large numbers, and furthermore we can do this for any batch size L.

Really the VAEs were just a straightforward application of CAVI with the reparamaterization trick, plus the extra variables α, β that are used to generate the θ, ϕ.

Algorithm 1.1 (Implementation of VAE)

To see an implementation of a VAE with PyTorch, see here.

2 Importance Weighted Autoencoders

3 Fisher Autoencoders

4 Conditional VAEs

Note that once our VAE is trained, we have no control on the data generation process. That is, if we want to generate only the digit 2, we can't since we must just sample from the Gaussian $p(z)$ in the latent space. Therefore, conditioning all the distributions on what we want is the objective of *conditional VAEs*, which uses the *conditional ELBO*.

Definition 4.1 (Conditional ELBO)

The **conditional ELBO** of x, θ, ϕ given some conditioning vector c[a], is defined

$$\mathbb{E}_{q_\phi(z|x)}\big[\log p_\theta(x \mid z, c) - \mathrm{KL}(q_\phi(z \mid x, c)||p(z \mid c))\big] \tag{47}$$

[a] Analogous to the digit we want in MNIST.

References

[KW22] Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2022.