Chapter 1

Generative modeling meets representation learning

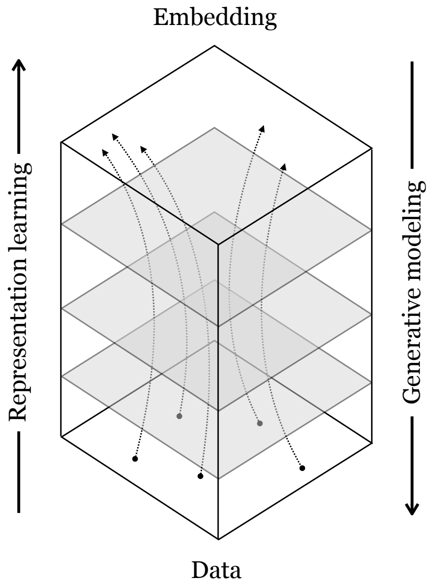
Draft chapter from Torralba, Isola, Freeman

This chapter is about models that unite the ideas of both generative modeling and representation learning. These models learn mappings both to and from data.

The intuition is that generative models map a simple base distribution ("noise") to data, whereas representation learning maps data to simple underlying representations ("embeddings"). These two problems are, essentially, inverses of each other. Many algorithms explicitly treat them as inverse problems, where solving the problem in one direction can inform the solution in the other direction.

In Chapter **??**, we described neural nets as being a sequence of mappings from raw data to ever more abstracted representations, layer by layer. This perspective puts representation learning in the spotlight: deep learning is just representation learning! Let us now point out an alternative perspective: in backwards order, deep nets are mappings from abstracted representations to ever more concrete representations of the data, layer by layer. This "backwards" ordering is the direction in which deep generative networks work. This perspective puts the spotlight on generative modeling: deep learning is just generative modeling! Indeed both modeling directions are valid, and the full picture looks like this:

Moving backwards through a net is also what backprop does, but it computes a different function: backprop computes the gradient ∇f whereas here we focus on the inverse f^{-1}.



Here we label one side as "Data" and the other as "Embedding", but what's the precise difference between these two things? Why is an RGB image "data" while a 100-dimensional vector of neural activations is an "embedding"? This is a question for you to think about; there is no "right" answer.

1.1 Latent variables as representations

In Chapter **??**, we introduced generative models with latent variables $\mathbf{z}$. In that context, the role of the latent variable was to specify all unobserved factors that might affect the output of a model. For example, if the model predicts the color of a black and white photo, it is a mapping $g : \mathbf{x}, \mathbf{z} \to \mathbf{y}$, with $\mathbf{x}$ being the black and white input, $\mathbf{y}$ being the color output and $\mathbf{z}$ being any other information that needs to be known in order to make the mapping completely deterministic – for example, the color of a t-shirt which cannot be inferred solely form the black and white input. In the extreme case of unconditional generative models, all properties of the generated images are controlled by the latent variables.

What we did not mention in Chapter **??**, but will focus on now, is that *latent variables are representations of the data*. In the case of an unconditional generative model, the latent variables are a *complete* representation of the data: all information in the data is represented in the latent variables.

Given this connection, this chapter will ask the question: are latent variables *good* representations of the data? And can they be combined with other representation learning algorithms?

1.2 Technical setting

We will consider random variables $\mathbf{z}$ and $\mathbf{x}$ related as follows, with g being deterministic generator (a.k.a. decoder) and f being a deterministic encoder:

$$\mathbf{x} \sim p_{\texttt{data}} \qquad\qquad \mathbf{z} \sim p_{\mathbf{z}} \tag{1.1}$$

$$\hat{\mathbf{z}} = f(\mathbf{x}) \qquad\qquad \hat{\mathbf{x}} = g(\mathbf{z}) \tag{1.2}$$

f and g will be trained so that $g \approx f^{-1}$ which means that $\hat{\mathbf{x}} \approx \mathbf{x}$ and $\hat{\mathbf{z}} \approx \mathbf{z}$. In Figure **??**, we sketched how representation learning maps from a data domain to a simple embedding space. We can now put that diagram side by side with the equivalent diagram for generative modeling. Notice again how they are just the same thing in opposite directions:

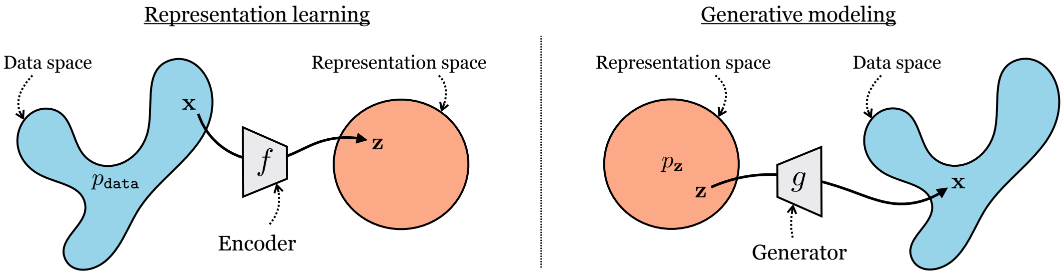


Figure 1.1: Generative modeling performs the opposite mapping from representation learning.

The critical thing in most generative models, which is not necessarily true for representation learning models, is that we assume we know the distribution $p_{\mathbf{z}}$, and typically it has a simple form such as a unit Gaussian. Knowing this distribution allows us to sample from it and then generate images via g.

One of the most important quantities we will measure is the data log likelihood function $L(\{\mathbf{x}\}_{i=1}^{N}, \theta)$, which measures the log likelihood of the data under the model p_θ:

$$L(\{\mathbf{x}^{(i)}\}_{i=1}^{N}, \theta) = \sum_{i=1}^{N} \log p_\theta(\mathbf{x}^{(i)}) \tag{1.3}$$

Many methods use a max likelihood objective, optimizing L w.r.t. θ. To compute the likelihood function, we need to compute $p_\theta(\mathbf{x})$. One way to express this function is as the **marginal likelihood** of $\mathbf{x}$, marginalizing over all unobserved latent variables $\mathbf{z}$:

$$p_\theta(\mathbf{x}) = \int_{\mathbf{z}} p_\theta(\mathbf{x}|\mathbf{z}) p_{\mathbf{z}}(\mathbf{z}) d\mathbf{z} \tag{1.4}$$

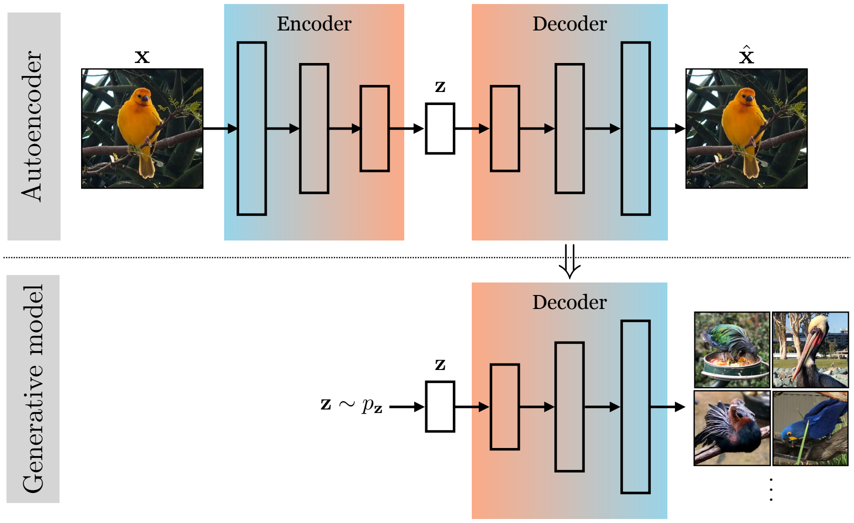
The advantage of expressing $p_\theta(\mathbf{x})$ in this way is that it reduces the modeling problem to learning the conditional distribution $p_\theta(X|\mathbf{z})$, which itself can be straightforwardly modeled using g (again, assuming we know $p_{\mathbf{z}}$). For example, we could model $p_\theta(X|\mathbf{z}) = \mathcal{N}(\mu = g(\mathbf{z}), \sigma = \mathbf{1})$, i.e . just place a unit Gaussian distribution centered on $g(\mathbf{z})$.

The integral in Eqn. 1.4 is expensive so most generative models either approximate the integral or somehow sidestep the need to explicitly calculate it. We will examine a few such strategies next.

1.3 Variational Autoencoders

1.3.1 The decoder of an autoencoder is a data generator

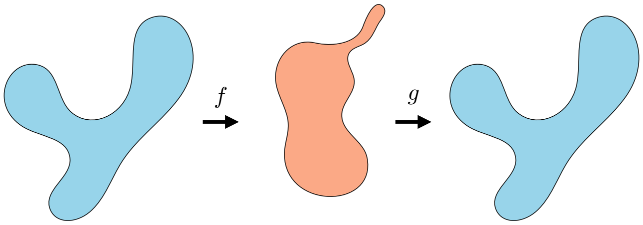
In Chapter **??** we learned about autoencoders. These are models that learn an embedding that can be decoded to reconstruct the input data. You may already have noticed that the decoder of an autoencoder looks just like a generator. It is a mapping from a representation of the data, $\mathbf{z}$, back to the data itself, $\mathbf{x}$. Given a $\mathbf{z}$, we can synthesize an image by passing it through the decoder of the autoencoder:



But how do we get this $\mathbf{z}$? One goal of generative modeling is to be able to make up random images from scratch. So we need a distribution from which we can sample $\mathbf{z}$'s from scratch, i.e. we need $p_{\mathbf{z}}$. An autoencoder doesn't directly give us this. You might ask, what if, after training an autoencoder, you just sample a random $\mathbf{z}$, say from a unit Gaussian, and feed it through the decoder? The problem is that this sample might be very different from what the decoder was trained on, and it therefore might not map to a natural looking image. For example, maybe the encoder has learned to map all images to embeddings far from the origin; then a unit Gaussian $\mathbf{z}$ would be far out of distribution and the decoder's behavior could be arbitrary for this out of distribution input.

In general, the embedding space of an autoencoder might be just as complicated to model as the data space we started with:

Variational autoencoders (**VAE**s) are a way to turn an autoencoder into a proper generative model, which can be sampled from and which maximizes data likelihood under



a formal probabilistic model. The trick is very simple: just take a vanilla autoencoder and 1) regularize the latent distribution to squish it into a Gaussian (or some other base distribution), 2) add noise to the output of the encoder. In code, it can be as simple as a two line change!

Okay, but seeing *why* this is the right and proper thing to do requires quite a bit of math. We will derive it now, using a different approach than in most texts. We think this approach makes it more intuitive what is going on. See [Kingma et al. 2019] for the more standard derivation.

1.3.2 The VAE hypothesis space

VAE's are max likelihood generative models, which maximize the likelihood function L in Eqn. 1.4. What distinguishes them from other max likelihood models is their particular hypothesis space and optimization algorithm. We will first describe the hypothesis space.

Remember the Gaussian generative model from Chapter **??** (a.k.a. fitting a Gaussian to data)? We stated that this model is too simple for most purposes, but can form the basis of more flexible density models, which work by combining a lot of simple distributions. We gave one example in Chapter **??**: autoregressive models, which model a complicated distribution as a product over many simple conditional distributions. VAEs follow a slightly different strategy: they model complicated distributions as *sums* of simple distributions.

Mixture models are probability models of the form $P(\mathbf{x}) = \sum_i w_i p_i(\mathbf{x})$.

In particular, VAEs are **mixture models**, and the most common kind of VAE is a **mixture of Gaussians**. The mixture of Gaussians model is in fact classical model that represents a density as a weighted sum of Gaussian distributions:

$$p_\theta(\mathbf{x}) = \sum_{i=1}^{k} w_i \mathcal{N}(\mathbf{x}; \mu_i, \mathbf{\Sigma}_i) \qquad \triangleleft \text{ Mixture of Gaussians} \tag{1.5}$$

where the parameters are $\theta = \{\mu_i, \mathbf{\Sigma}_i\}_{i=1}^{k}$, i.e. the mean and variance of all Gaussians in the mixture. Unlike classical Gaussian mixture models, VAEs use an *infinite* mixture of Gaussians, i.e. $k \rightarrow \infty$.

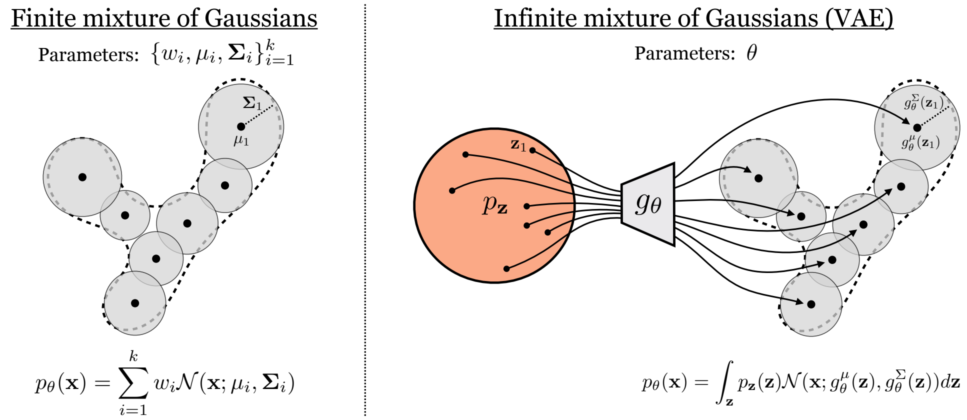
You can think of a function as the *infinite set* of values a variable takes on over some domain.

But wait, how can we parameterize an infinite mixture? We can't learn an infinite set of means and variances. The trick we will use is to make the mean and variance be *functions* of an underlying continuous variable. The function the VAE uses is g_θ. For notational convenience, we decompose this function into g_θ^μ and g_θ^Σ to separately model the means and variances of the Gaussians in the infinite mixture. Next we need a continuous domain to integrate our mixture over, and, as a simple choice, we will use the unit Gaussian distribution. Then our infinite mixture can be described as:

$$p_\theta(\mathbf{x}) = \int_{\mathbf{z}} \underbrace{\mathcal{N}(\mathbf{x}; g_\theta^\mu(\mathbf{z}), g_\theta^\Sigma(\mathbf{z}))}_{p_\theta(\mathbf{x}|\mathbf{z})} \underbrace{\mathcal{N}(\mathbf{z}; \mathbf{0}, \mathbf{I})}_{p_{\mathbf{z}}(\mathbf{z})} d\mathbf{z} \qquad \triangleleft \text{ VAE hypothesis space} \tag{1.6}$$

Notice that this equation – an infinite mixture of Gaussians parameterized by a function g_θ – has exactly the same form as the marginal likelihood Eqn. 1.4. What we have done is model an infinite mixture as an integral that marginalizes over a continuous latent variable.

You can think about this as transforming a base distribution $p_{\mathbf{z}}$ to a modeled distribution p_θ by applying a deterministic mapping g_θ and then putting a blip of Gaussian probability around each point in the range of this mapping. If you sample a few of the Gaussians in this infinite mixture, they might look like this:


Finite mixture of Gaussians
Parameters: $\{w_i, \mu_i, \mathbf{\Sigma}_i\}_{i=1}^k$
$\mathbf{\Sigma}_1$
μ_1
$p_\theta(\mathbf{x}) = \sum_{i=1}^k w_i \mathcal{N}(\mathbf{x}; \mu_i, \mathbf{\Sigma}_i)$
Infinite mixture of Gaussians (VAE)
Parameters: θ
$\mathbf{z}_1$
$p_\mathbf{z}$
g_θ
$g_\theta^\Sigma(\mathbf{z}_1)$
$g_\theta^\mu(\mathbf{z}_1)$
$p_\theta(\mathbf{x}) = \int_\mathbf{z} p_\mathbf{z}(\mathbf{z})\mathcal{N}(\mathbf{x}; g_\theta^\mu(\mathbf{z}), g_\theta^\Sigma(\mathbf{z}))d\mathbf{z}$

While we have chosen Gaussians for $p_\theta(\mathbf{x}|\mathbf{z})$ and $p_\mathbf{z}(\mathbf{z})$ in this example, VAEs can also be constructed using other base distributions, even complicated ones. For example, we could make an infinite mixture of the autoregressive distributions we saw in Chapter **??**. In this sense, mixture models are meta-models, and their components can themselves be any of the density models we have learned about in this book, including other mixture models.

1.3.3 Optimizing VAEs

With the objective and hypothesis given above, we can now fully define the VAE learning problem:

Wait, a whole section on optimization? Didn't this book say that general-purpose optimizers (like backprop) are sufficient in the deep learning era? Yes. *But only if you can actually compute the objective and its gradient.* The issue here is that the VAE's objective is *intractable.* Its exact computation requires integrating over an infinite set of deep net forward passes. The difficulty of optimizing VAEs lies in the difficulty of approximating this intractable objective. Once we derive our approximation, optimization again will be easy: just apply backprop on this approximate loss.

$$\theta^* = \arg\min_\theta L(\{\mathbf{x}^{(i)}\}_{i=1}^N, \theta) \tag{1.7}$$

$$= \arg\min_\theta \sum_{i=1}^N \log \underbrace{\int_\mathbf{z} \overbrace{\mathcal{N}(\mathbf{x}^{(i)}; g_\theta^\mu(\mathbf{z}), g_\theta^\Sigma(\mathbf{z}))}^{p_\theta(\mathbf{x}^{(i)}|\mathbf{z})} \overbrace{\mathcal{N}(\mathbf{z}; \mathbf{0}, \mathbf{I})}^{p_\mathbf{z}(\mathbf{z})} \, d\mathbf{z}}_{p_\theta(\mathbf{x}^{(i)})} \tag{1.8}$$

Trick #1: Approximating the objective via sampling The integral for $p_\theta(\mathbf{x}^{(i)})$ does not necessarily have a closed form since g_θ may be an arbitrarily complex function. Therefore, we need to approximate this integral numerically. The first trick we will use is a *Monte Carlo estimate* of this integral:

$$p_\theta(\mathbf{x}) = \int_\mathbf{z} p_\theta(\mathbf{x}|\mathbf{z}) p_\mathbf{z}(\mathbf{z}) d\mathbf{z} \tag{1.9}$$

$$= \mathbb{E}_{\mathbf{z}\sim p_\mathbf{z}(\mathbf{z})}[p_\theta(\mathbf{x}|\mathbf{z})] \tag{1.10}$$

$$\approx \frac{1}{M}\sum_{i=1}^M p_\theta(\mathbf{x}|\mathbf{z}^{(j)}), \quad \mathbf{z}^{(j)} \sim p_\mathbf{z} \tag{1.11}$$

We could stop here, as the learning problem is now written in a closed differentiable form:

$$\theta^* = \arg\min_\theta \frac{1}{M}\sum_{i=1}^N \log \sum_{j=1}^M \overbrace{\mathcal{N}(\mathbf{x}^{(i)}; g_\theta^\mu(\mathbf{z}^{(j)}), g_\theta^\Sigma(\mathbf{z}^{(j)}))}^{p_\theta(\mathbf{x}^{(i)}|\mathbf{z}_j))} \tag{1.12}$$

As long as g_θ is a differentiable neural net, we can proceed with optimization via backprop. In practice, on each iteration of backprop, we would collect a batch of random samples

$\{\mathbf{x}^{(i)}\}_{i=1}^{B_1} \sim p_{\texttt{data}}$ from the training set and a batch of random latents $\{\mathbf{z}^{(j)}\}_{j=1}^{B_2} \sim p_{\mathbf{z}}$ from the unit Gaussian distribution (with B_1 and B_2 being batch sizes). Then we would pass each of the $\mathbf{z}$ samples through our net g_θ, which yields a Gaussian under which we can evaluate each $\mathbf{x}$ sample. After evaluating and summing up the log probabilities, we would run a backwards pass to update the parameters θ.

Below we show what this process looks like at three checkpoints during training. Here we use an isotropic Gaussian model, i.e. we parameterize the covariance as $\Sigma = \sigma\mathbf{I}$, where $\sigma = g_\theta^\sigma(\mathbf{z})$ is a scalar.

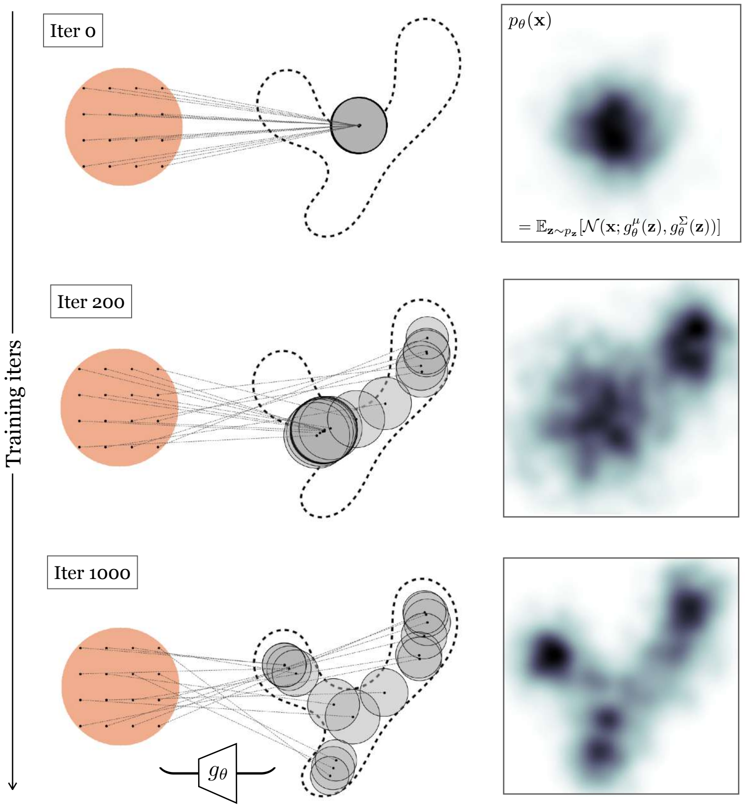

Iter 0
$p_\theta(\mathbf{x})$
$= \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}}[\mathcal{N}(\mathbf{x}; g_\theta^\mu(\mathbf{z}), g_\theta^\Sigma(\mathbf{z}))]$
Iter 200
Training iters
Iter 1000
g_θ

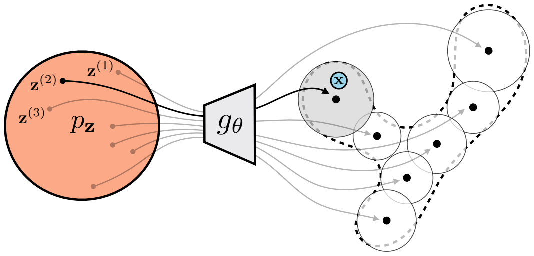
Figure 1.2: Fitting an infinite mixture of Gaussians whose means and variances are parameterized by a generator function g_θ.

Trick #2: Efficient approximation via importance sampling The above works decently for modeling low-dimensional distributions. Unfortunately, this approach does not scale well to high-dimensions. The reason is that in order for our Monte Carlo estimate of the integral to be accurate, we may need many samples from $p_{\mathbf{z}}$, and the higher the dimensionality of $\mathbf{z}$, the more samples we will typically need.

Can we come up with a more efficient way of approximating the integral in Eqn. 1.9? Let's start by writing out the sum from Eqn. 1.11 more explicitly:

$$p_\theta(\mathbf{x}) \approx \frac{1}{M}(p_\theta(\mathbf{x}|\mathbf{z}^{(1)}) + p_\theta(\mathbf{x}|\mathbf{z}^{(2)}) + p_\theta(\mathbf{x}|\mathbf{z}^{(3)}) + \ldots) \tag{1.13}$$

In general, some of the terms $p_\theta(\mathbf{x}|\mathbf{z}^{(j)})$ will be larger than others. In fact, in our example in Figure **??**, *most* of these terms are near zero. This is because, to maximize likelihood, the model spread out the Gaussians so that each places high density on a different part of the data distribution. A datapoint $\mathbf{x}$ will only have substantial probability under the Gaussians whose means are near $\mathbf{x}$. Consider the example below, where we are trying to esimate the probability of the point $\mathbf{x}$ (blue circle):



The mixture components are shaded according to the probability they assign to $\mathbf{x}$. Almost all are so far from $\mathbf{x}$ that we have:

$$p_\theta(\mathbf{x}) \approx \frac{1}{M}(0 + p_\theta(\mathbf{x}|\mathbf{z}^{(2)}) + 0 + \ldots) \tag{1.14}$$

If we had *only* sampled $\mathbf{z}^{(2)}$, we would have had almost as good an estimate!

This brings us to the second trick of VAE's: when approximating the likelihood integral for $p_\theta(\mathbf{x})$, try to only sample $\mathbf{z}$'s that place high probability on $\mathbf{x}$, i.e. those $\mathbf{z'}s$ for which $p_\theta(\mathbf{x}|\mathbf{z})$ is large. Then, a few samples will suffice to well approximate the entire expectation. This trick is called **importance sampling**. It is a general trick for approximating expectations. Rather than sampling $\mathbf{z}^{(i)} \sim p_{\mathbf{z}}$, we sample from some other density $\mathbf{z}^{(i)} \sim q_{\mathbf{z}}$, and multiply by a correction factor $\frac{p_{\mathbf{z}}(\mathbf{z})}{q_{\mathbf{z}}(\mathbf{z})}$ to account for the fact that we are sampling from a biased distribution:

$$p_\theta(\mathbf{x}) = \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}}\Big[p_\theta(\mathbf{x}|\mathbf{z})\Big] = \int_{\mathbf{z}} p_{\mathbf{z}}(\mathbf{z})p_\theta(\mathbf{x}|\mathbf{z})d\mathbf{z} = \int_{\mathbf{z}} q_{\mathbf{z}}(\mathbf{z})\frac{p_{\mathbf{z}}(\mathbf{z})}{q_{\mathbf{z}}(\mathbf{z})}p_\theta(\mathbf{x}|\mathbf{z})d\mathbf{z} \tag{1.15}$$

$$= \mathbb{E}_{\mathbf{z}\sim q_{\mathbf{z}}}\Big[\frac{p_{\mathbf{z}}(\mathbf{z})}{q_{\mathbf{z}}(\mathbf{z})}p_\theta(\mathbf{x}|\mathbf{z})\Big] \tag{1.16}$$

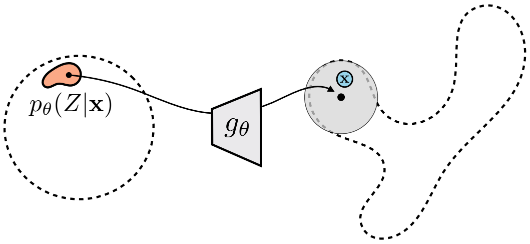
Using the intuition we developed above, the distribution q we would really like to sample from is the one whose samples maximize $p_\theta(\mathbf{x}|\mathbf{z})$. It turns out that the optimal distribution is $q^* = p_\theta(Z|\mathbf{x})$. This distribution minimizes the expected error between a Monte Carlo estimate of the expectation and its true value (i.e. minimizes the variance of our Monte Carlo estimator). The intuition is that $p_\theta(Z|\mathbf{x})$ is precisely a prediction of which $\mathbf{z}$'s are most likely to have generated the observed $\mathbf{x}$. The optimal importance sampling way to estimate the likelihood of a datapoint will therefore look like this:

See [Owen 2013], Chapter 9.1 for a proof that $q^*(\mathbf{z}) \propto |p_\theta(\mathbf{x}|\mathbf{z})|p_{\mathbf{z}}(\mathbf{z})$, from which it then follows that $q^*(\mathbf{z}) \propto p_\theta(\mathbf{x}|\mathbf{z})p_{\mathbf{z}}(\mathbf{z}) = p_\theta(\mathbf{x},\mathbf{z}) \propto p_\theta(\mathbf{z}|\mathbf{x})$, yielding our result

$$p_\theta(\mathbf{x}) \approx \frac{1}{M}\sum_{j=1}^{M}\frac{p_{\mathbf{z}}(\mathbf{z}^{(j)})}{p_\theta(\mathbf{z}^{(j)}|\mathbf{x})}p_\theta(\mathbf{x}|\mathbf{z}^{(j)}) \qquad \triangleleft \text{ Optimal importance sampling} \tag{1.17}$$

$$\mathbf{z}^{(j)} \sim p_\theta(Z|\mathbf{x})$$

Visually, rather than sampling from all over $p_{\mathbf{z}}$ and wasting samples on regions that places nearly zero likelihood on the data, we focus our sampling to just the region that places high likelihood on the data, and can get then away with far fewer samples to well approximate the data likelihood:

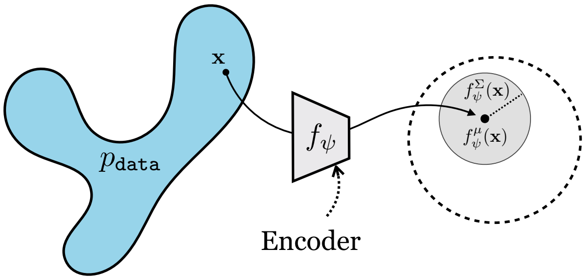


Trick #3: Variational inference to approximate the sampling distribution Now we know what distribution we *should* be sampling $\mathbf{z}$'s from: $p_\theta(Z|\mathbf{x})$. The only remaining problem is that this distribution may be complicated and hard to sample from. Sampling from arbitrary distributions is a standard topic in statistics, and many algorithms have been proposed, including the MCMC methods we encountered in previous chapters. VAE's use a strategy called **variational inference**.

Remember, we have defined simple forms for only $p_\theta(X|\mathbf{z})$ and $p_\mathbf{z}$ – both are Gaussians – but this does not mean $p_\theta(Z|\mathbf{x})$ has a simple form.

The name "variational" comes from the "calculus of variations", which studies functionals (functions of functions). Integrals of probability densities are functionals. Variational inference is commonly (but not always) used to approximate densities expressed as the integral of some other densities, hence functionals, hence the name "variational".

The idea of variational inference is to approximate an intractable density p by finding the nearest density in a tractable family q_ψ, parameterized by ψ. In VAE's, we approximate our ideal importance density $p_\theta(Z|\mathbf{x})$ with a q_ψ in a family we can efficiently sample from – the most common choice is to again use a Gaussian, conditioned on $\mathbf{x}$: $q_\psi = \mathcal{N}(f^\mu_\psi(\mathbf{x}), f^\Sigma_\psi(\mathbf{x}))$. f_ψ is a function that maps from $\mathbf{x}$ to parameters of a distribution over $\mathbf{z}$ – in other words, f_ψ is a probabilistic encoder!



It will turn out that f indeed plays the role of the encoder in the VAE, while g plays the role of the decoder.

We want q_ψ to be the best approximation of $p_\theta(Z|\mathbf{x})$, so our goal will be to choose the q_ψ that minimizes the KL-divergence between these two distributions (we could have chosen other divergences or notions of approximation, but we will see that using KL yields some nice properties). We will call the objective for q_ψ as J_q and will rewrite it as follows:

$$\begin{aligned}
J_q &= \texttt{KL}(q_\psi(Z|\mathbf{x}) \parallel p_\theta(Z|\mathbf{x})) && (1.18)\\
&= \mathbb{E}_{\mathbf{z}\sim q_\psi(Z|\mathbf{x})}[\log q_\psi(\mathbf{z}|\mathbf{x}) - \log p_\theta(\mathbf{z}|\mathbf{x})] && (1.19)\\
&= \mathbb{E}_{\mathbf{z}\sim q_\psi(Z|\mathbf{x})}[\log q_\psi(\mathbf{z}|\mathbf{x}) - \log p_\theta(\mathbf{x}|\mathbf{z}) + \log p_\mathbf{z}(\mathbf{z}) - \log p_\theta(\mathbf{x})] && (1.20)\\
&= \mathbb{E}_{\mathbf{z}\sim q_\psi(Z|\mathbf{x})}[\log q_\psi(\mathbf{z}|\mathbf{x}) - \log p_\theta(\mathbf{x}|\mathbf{z}) + \log p_\mathbf{z}(\mathbf{z})] - \log p_\theta(\mathbf{x}) && (1.21)
\end{aligned}$$

The last line follows from the fact that $\log p_\theta(\mathbf{x})$ is a constant w.r.t. the distribution we are taking expectation over.

The learning problem for q_ψ is to optimize J_q w.r.t. parameters ψ. Notice that $\log p_\theta(\mathbf{x})$ is constant w.r.t. these parameters, so we have that:

$$\begin{aligned}
\psi^* &= \arg\min_\psi J_q(\psi) && (1.22)\\
&= \arg\min_\psi \underbrace{\mathbb{E}_{\mathbf{z}\sim q_\psi(Z|\mathbf{x})}[\log q_\psi(\mathbf{z}|\mathbf{x}) - \log p_\theta(\mathbf{x}|\mathbf{z}) + \log p_\mathbf{z}(\mathbf{z})]}_{J} && (1.23)
\end{aligned}$$

Here we have defined a new cost function, J, whose minimizer w.r.t. ψ is the same as the minimizer for J_q.

Now, let us now recall our learning problem for p_θ, which is to maximize data log likelihood. Using importance sampling to estimate data likelihood (Eqn. 1.16), and using q_ψ as our sampling distribution, we have that the objective for p_θ is to minimize the following cost J_p w.r.t. θ:

$$\begin{aligned}
J_p &= -\log \mathbb{E}_{\mathbf{z}\sim q_\psi(Z|\mathbf{x})}\Big[\frac{p_\mathbf{z}(\mathbf{z})}{q_\psi(\mathbf{z}|\mathbf{x})} p_\theta(\mathbf{x}|\mathbf{z})\Big] && (1.24)\\
\theta^* &= \arg\min_\theta J_p(\theta) && (1.25)
\end{aligned}$$

We now have a differentiable objective for both ψ and θ; the objective for ψ is expressed as an expectation and can be optimized by taking a Monte Carlo sample from that expectation. We *could* also try using Monte Carlo to approximate the objective for θ, but this would yield a biased esimator of θ, since Equation 1.24 has the log outside the expectation. That might be okay (as the number of samples N goes to infinity, the bias goes to zero), but we can do better. To get around this issue, VAEs adopt the following strategy: rather than minimizing J_p w.r.t. p_θ, they minimize an upper-bound to J_p, which is expressed purely as an expectation and yields unbiased Monte Carlo estimates. The particular upper-bound used is in fact J – the same objective as we used for optimizing ψ in Eqn. 1.23! The fact that J is a upper-bound on J_p follows from Jensen's inequality:

$\log \frac{1}{N}\sum_i x_i$ is not an unbiased estimator of $\log \mathbb{E}[x]$, hence a Monte Carlo estimate is not the best choice for Equation 1.24.

$$\begin{aligned}
J_p &= -\log \mathbb{E}_{\mathbf{z}\sim q_\psi(Z|\mathbf{x})}\Big[\frac{p_\mathbf{z}(\mathbf{z})}{q_\psi(\mathbf{z}|\mathbf{x})}p_\theta(\mathbf{x}|\mathbf{z})\Big] && (1.26)\\
&\leq -\mathbb{E}_{\mathbf{z}\sim q_\psi(Z|\mathbf{x})}\Big[\log(\frac{p_\mathbf{z}(\mathbf{z})}{q_\psi(\mathbf{z}|\mathbf{x})}p_\theta(\mathbf{x}|\mathbf{z}))\Big] \qquad \triangleleft \text{ Jensen's inequality} && (1.27)\\
&= \mathbb{E}_{\mathbf{z}\sim q_\psi(Z|\mathbf{x})}\Big[\log q_\psi(\mathbf{z}|\mathbf{x}) - \log p_\theta(\mathbf{x}|\mathbf{z}) - \log p_\mathbf{z}(\mathbf{z})\Big] && (1.28)\\
&= J && (1.29)
\end{aligned}$$

We can actually be more precise about the gap between J_p and J: $J - J_p = \mathtt{KL}(q_\psi(Z|\mathbf{x}) \parallel p_\mathbf{z})$, which is strictly non-negative (exercise: do the algebra to show this). $-J$ is often referred to as the **Evidence Lower Bound** or **ELBO**.

This way our learning problem for both θ and ψ share the same objective (which saves computation) and can be stated simply as:

$$\theta^*, \psi^* = \arg\min_{\theta,\phi} J(\theta, \phi) \tag{1.30}$$

Connection to autoencoders You may have noticed that in the previous sections we made use of both an encoder f_ψ (which parameterizes $q_\psi(Z|\mathbf{x})$) and a decoder g_θ (which parameterizes $p_\theta(X|\mathbf{z})$) – it looks like we are using the two pieces of an autoencoder but what's the exact connection? To see it, we need to write out the objective J in more explicit detail. We will derive the connection for a simple VAE on 1D data with 1D latent space, i.e. $x \in \mathbb{R}$, $z \in \mathbb{R}$.

$$\begin{aligned}
J &= \mathbb{E}_{z\sim q_\psi(Z|x)}\Big[\log q_\psi(z|x) - \log p_\theta(x|z) - \log p_z(z)\Big] && (1.31)\\
&= \mathbb{E}_{z\sim q_\psi(Z|x)}\Big[-\log p_\theta(x|z)\Big] + \mathtt{KL}(q_\psi(Z|x) \parallel p_z(z)) && (1.32)\\
&= \mathbb{E}_{z\sim q_\psi(Z|x)}\Big[-\log \mathcal{N}(x; g^\mu_\theta(z), g^\sigma_\theta(z))\Big] - \mathtt{KL}(q_\psi(Z|x) \parallel \mathcal{N}(0,1)) && (1.33)
\end{aligned}$$

Computing this expectation requires drawing samples from $q_\psi(Z|x)$. We can draw such a sample as $z = f^\mu_\psi(x) + \epsilon f^\sigma_\psi(x)$ with $\epsilon \sim \mathcal{N}(0,1)$. This allows rewriting the expectation as:

This is known in the literature as the **reparameterization trick**

$$\begin{aligned}
J &= \mathbb{E}_{\epsilon\sim\mathcal{N}(0,1)}\Big[-\log \mathcal{N}(x; g^\mu_\theta(z), g^\sigma_\theta(z))\Big] - \mathtt{KL}(q_\psi(Z|x) \parallel \mathcal{N}(0,1)) && (1.34)\\
&= \mathbb{E}_{\epsilon\sim\mathcal{N}(0,1)}\Big[-\log \mathcal{N}(x; g^\mu_\theta(f^\mu_\psi(x) + \epsilon f^\sigma_\psi(x)), g^\sigma_\theta(f^\mu_\psi(x) + \epsilon f^\sigma_\psi(x)))\Big]\\
&\qquad - \mathtt{KL}(q_\psi(Z|\mathbf{x}) \parallel \mathcal{N}(0,1)) && (1.35)\\
&= \mathbb{E}_{\epsilon\sim\mathcal{N}(0,1)}\Big[\log\frac{1}{\sigma\sqrt{2\pi}} - \frac{\overbrace{(x-\mu)^2}^{\text{reconstruction error}}}{2\sigma^2}\Big] - \mathtt{KL}(q_\psi(Z|\mathbf{x}) \parallel \mathcal{N}(0,1)) && (1.36)\\
&\mu = \underbrace{g^\mu_\theta(f^\mu_\psi(x) + \epsilon f^\sigma_\psi(x)),}_{\text{noisy autoencoder}} \qquad \sigma = g^\sigma_\theta(f^\mu_\psi(x) + \epsilon f^\sigma_\psi(x)))
\end{aligned}$$

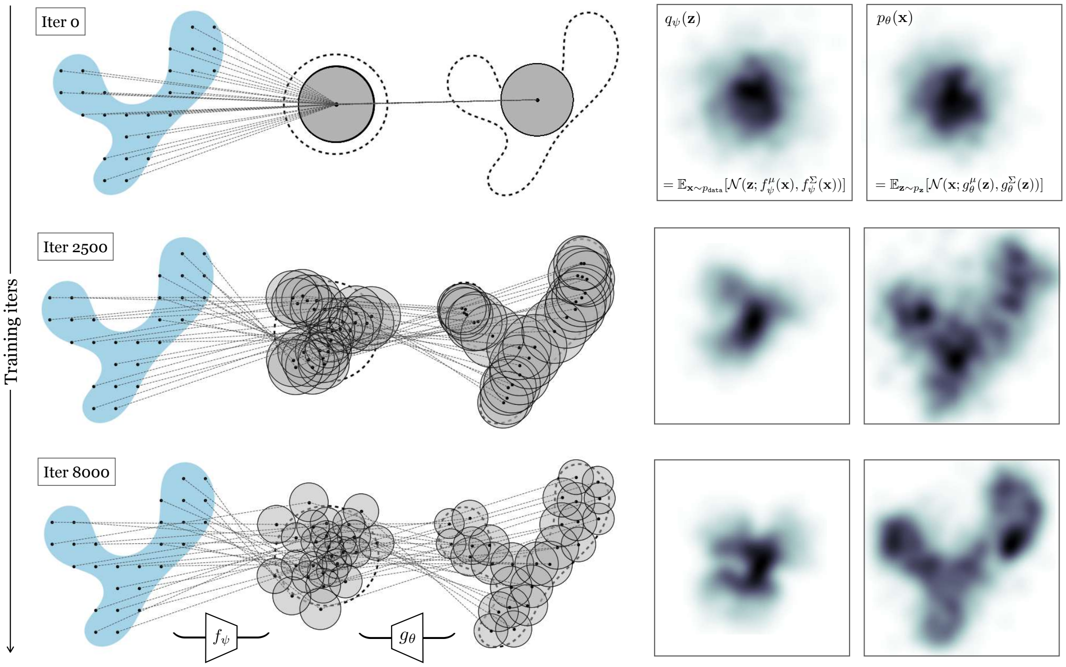
What this shows is that to compute the VAE's objective, you simply run an autoencoder with noise added to the bottleneck. The cost function is then the reconstruction error (same as with regular autoencoders) weighted by a term related to the variance of the noise added and also with an additional KL-divergence term to squish the latent distribution toward the unit Gaussian.

To train a VAE we normally are not just fitting a single $\mathbf{x}$ but maximizing the likelihood of a dataset $\{\mathbf{x}^{(i)}\}_{i=1}^{N}$, so the cost we optimize is $\sum_i J(\mathbf{x}^{(i)}, \theta, \phi)$. Optimization of VAE proceeds as follows:

- Sample one or more $\mathbf{x} \sim \{\mathbf{x}^{(i)}\}_{i=1}^{N}$
- *Encode* the data with a forward pass through f_ψ
- For each datapoint, create one or more noisy latent codes using the distribution parameterized by the encoder
- *Decode* the data by passing the noisy latent codes through the g_θ
- Compute the losses and backprop to update θ and ψ

In most implementations of VAEs, only a single sample from the latent distribution is used for each datapoint on each iteration of backprop.

Three checkpoints of optimizing a VAE are shown below. As in the infinite mixture of Gaussians example above, we again assume an isotropic Gaussian model for the decoder, and here also assume that model for the encoder:

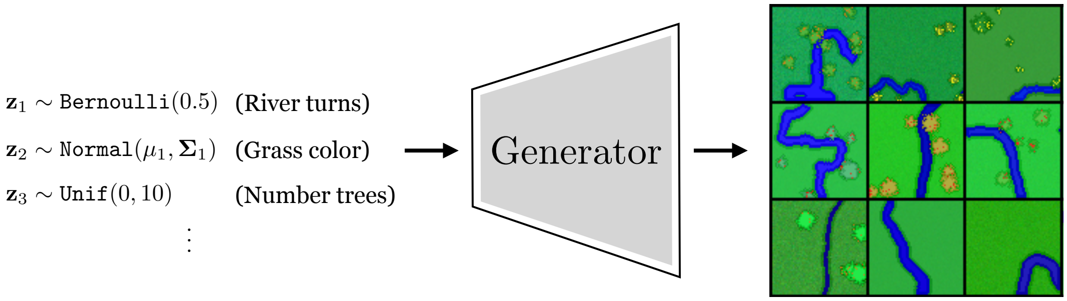

Iter 0
Iter 2500
Iter 8000
Training iters
f_ψ
g_θ
$q_\psi(\mathbf{z})$
$= \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}[\mathcal{N}(\mathbf{z}; f^{\mu}_{\psi}(\mathbf{x}), f^{\Sigma}_{\psi}(\mathbf{x}))]$
$p_\theta(\mathbf{x})$
$= \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}}[\mathcal{N}(\mathbf{x}; g^{\mu}_{\theta}(\mathbf{z}), g^{\Sigma}_{\theta}(\mathbf{z}))]$

1.3.4 Do VAEs learn good representations?

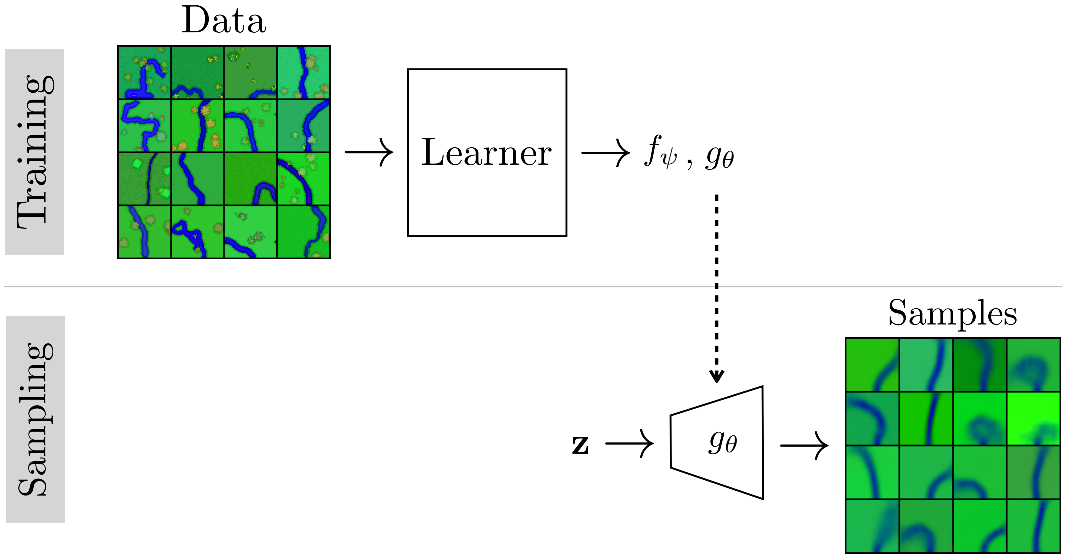
One perspective on VAEs is that they are a way to train a generative model p_θ. From this perspective, the encoder is just scaffolding for learning a decoder. However, the encoder can also be useful as an end in itself, and we might instead think of the decoder as scaffolding for training an encoder. This was the perspective presented by autoencoders, after all, and the VAE encoder comes with the same useful properties: it maps data to a low-dimensional latent code that preserves information about the input. In fact, from a representation learning perspective, VAE's even go beyond autoencoders. Not only do VAE's learn compressed embedding, the embeddings may also have other desirable properties depending on the prior $p_{\mathbf{z}}$. For example, if $p_{\mathbf{z}} = \mathcal{N}(0, 1)$, as is common, then the loss encourages that the dimensions of the embedding are independent, a property called **disentanglement**.

Disentanglement means that we can vary one dimension of the embedding at time, and just one independent factor of variation in the generated images will change. For example, one dimension might control the direction of light in a scene and another dimension might control the intensity of light.

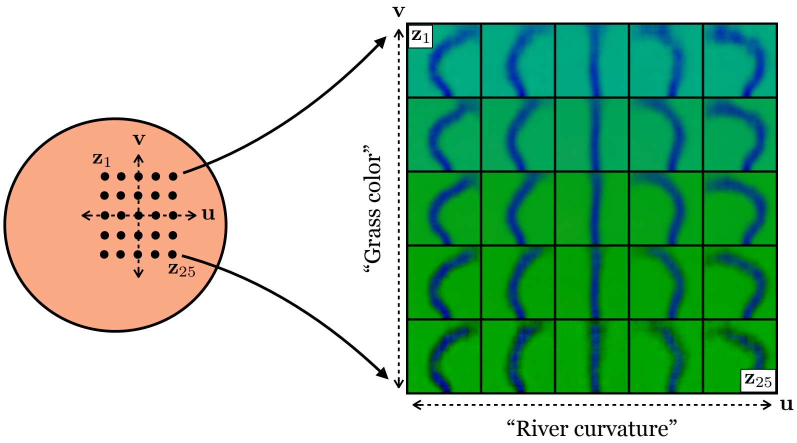
Example: learning a VAE for rivers Suppose we have a dataset of aerial views of rivers. We wish to fit this data with a VAE so that 1) we can generate new rivers and 2) we can identify the underlying latent variables that explain river appearance. In this example we will use data for which we know the true data generating process – we will use a python script that procedurally synthesizes cartoon images of rivers given input noise (this is a more elaborate version of the script we saw in Algorithm **??** of Chapter **??**). The script takes in random values that control the attributes of the scene – the grass color, the heading of the river, the number of trees, etc – and generates an image with these attributes:


$\mathbf{z}_1 \sim \texttt{Bernoulli}(0.5)$ (River turns)
$\mathbf{z}_2 \sim \texttt{Normal}(\mu_1, \mathbf{\Sigma}_1)$ (Grass color)
$\mathbf{z}_3 \sim \texttt{Unif}(0, 10)$ (Number trees)
$\vdots$
Generator

Training a VAE on this data learns to recreate the generative process with a *neural net* (rather than a python script) and maps zero-mean unit-variance *Gaussian noise* to images (rather than taking as input the noise types the script uses):



Did the VAE uncover the “true” latent variables that generated the data, i.e. did it recover latent dimensions corresponding to the attribute values that were the inputs to the python script? We can examine this by generating a set of images that walk along two latent dimensions of the VAE's z-space:



One of the latent dimensions seems to control the grass color, and another controls the river curvature! These two latent dimensions are disentangled in the sense that varying the latent dimension that controls color has little effect on curvature and varying the latent dimension that controls curvature has little effect on color. Indeed, grass color was one of the attributes of the true data generating process (the python script), and the VAE recovered it. Interestingly, however, there was no single input to the script that controls the overall river curvature, instead the curves are generating by a vector of Bernoulli variables that rotate the heading left and right as the river extends (using the same algorithm as in Algorithm **??** of Chapter **??**). The VAE has discovered a latent dimension that somehow summarizes a more global mode of behavior – bend left or bend right – than is explicit in the python script. It is important to realize that VAE's, and most representation learning methods, do not necessarily recover the true causal mechanisms that generated the data but rather might find other mechanisms that can equivalently explain the data.

A formal name for this issue is the **non-identifiability** of the true parameters that generated a dataset.

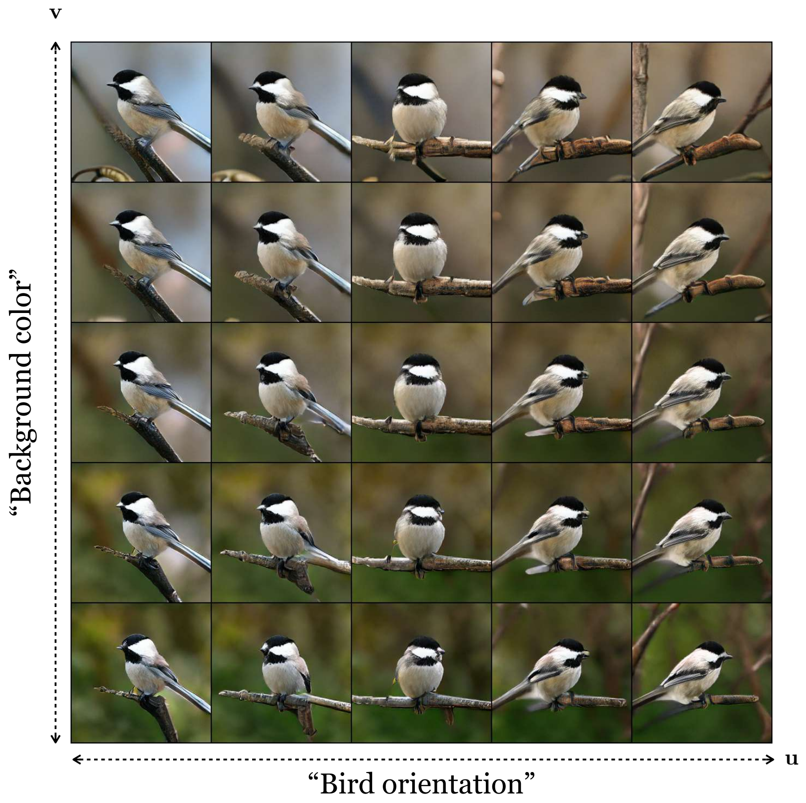
To summarize this section, we have seen that a VAE can be considered two things:

- An efficient way to optimize an infinite mixture of Gaussians generative model.
- A way to learn a low-dimensional, disentangled representation that can reconstruct the data.

1.4 Generative adversarial networks are representation learners

In Chapter **??** we covered GANs, which, like VAEs, train a mapping from latent variables to synthesized data, $g : \mathbf{z} \rightarrow \mathbf{y}$. Do GANs also learn meaningful and disentangled latents?

To see, let us repeat the experiment of examining latent dimensions of a generative model, but this time with GANs. Here we will use a powerful GAN, called BigGAN [Brock et al. 2019], that has been trained on the ImageNet dataset [Russakovsky et al. 2015]. Here are images generated by walking along two latent dimensions of this GAN:



Just like with the VAE trained on cartoon rivers, the GAN has also discovered disentangled latent variables; in this case they seem to control background color and the bird's orientation.

This makes sense: structurally, the GAN generator is very similar to the VAE decoder. In both cases, they map a low-dimensional random variable $\mathbf{z}$ to data, and typically $p_{\mathbf{z}} \sim \mathcal{N}(0, 1)$. That means that the dimensions of $\mathbf{z}$ are a priori independent (disentangled). In both models the goal is roughly the same: create synthetic data is has all the properties of real data. It should therefore come as no surprise that both models learn latent representations with similar properties. Indeed, these are just two examples of a large class of models that map low-dimensional latents from a simple (high entropy) distribution to high-dimensional data from a more structured (low entropy) distribution, and we might expect all models in this family to lead to similarly useful representations of the data.

Bibliography

A. Brock, J. Donahue, and K. Simonyan. Large scale gan training for high fidelity natural image synthesis. *International Conference on Learning Representations*, 2019.

D. P. Kingma, M. Welling, et al. An introduction to variational autoencoders. *Foundations and Trends® in Machine Learning*, 12(4):307–392, 2019.

A. B. Owen. *Monte Carlo theory, methods and examples.* 2013.

O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.