[SQUEAKING]

[RUSTLING]

[CLICKING]

**PHILLIP ISOLA:** We're going to continue on today and this week with generative models. So today will be on variational autoencoders. And we're going to give a perspective that generative modeling and representation learning are very tightly coupled. If you can solve the generative modeling problem, it should help you solve the representation learning problem and vice versa. And variational autoencoders are going to be one model that really puts the two ideas together and solves them both at the same time.

And this lecture is a little bit different than some of the last few, in that rather than covering a lot of different topics in some superficial level of detail, we're going to just go deep on VAEs, on variational autoencoders. And we'll also talk a little bit about the types of representations that are learned by generative models at the end and this idea of disentanglement.

So this is a picture I've shown a few different times throughout all of the representation learning and generative modeling lectures in this class. It's how I like to think about the two problems, where representation learning is going from raw data to some nicer, structured, abstracted representation. And generative modeling is going from a nice, simple, abstracted representation to data.

And I've said that these two things can be kind of thought of as inverses to some degree, not necessarily inverses in the sense of a mathematical function inverse because the direction from data to embeddings is many to one, and the direction from embeddings to data might be one to many. It might be stochastic, so it's more like a stochastic inverse, but they kind of operate in opposite directions.

And in this lecture, we're going to see a model that explicitly models both directions simultaneously. So representation learning mapping data to abstract representations, generative modeling, mapping abstract representations, like a text instruction for example to data, so make an image from some text.

So the perspective that we're going to look at today is going to be generative models as transformations from one probability distribution to another probability distribution. So we'll have a prior distribution, which will be the noise that we talked about last class. So the prior distribution is usually something that is very simple in structure. And we're going to map that prior distribution to a complicated data distribution.

So we'll always have our latent variables or our noise space z, which is often going to be Gaussian, indicated by a cartoon here as a circle because it's something that is meant to be simple. If it's actually a normal distribution, then in high dimensions, that will be roughly a soap bubble. That's the way to think of it. Most of the probability density of a Gaussian in high dimensions is right near the surface of the hypersphere, and the data distribution will be something more complicated. And so we're going to find a mapping from data that lives in this space.

And according to this distribution, we're going to geometrically push and pull it around, which is what the layers of a neural net do until we get it into this shape. So remember those diagrams I showed before of the distribution of input data points, going layer by layer and getting remapped into a new distribution. This is like a cartoon version of that diagram.

So data is x. Latent variables are z. And in the representation learning lectures, we looked at this direction-- I think I used the same cartoons. So we had a mapping f, which we called the encoder from data to representations. And in the generative modeling direction, it will be a mapping from representations to data. And we'll call the function that generates the data a generator g. So g is a mapping from z to x And f is a mapping from x to z. So these are sometimes called generators. Other times, they're called decoders. So we have an encoder that maps data to representation a decoder that maps representation to data.

So we've seen that a few times. Now let's put these two things together. It looks like they're basically solving related problems. And one model that puts these two things together is, of course, the autoencoder. Maybe you already kind of noticed that. So an autoencoder maps from data to some vector space, low-dimensional representation z. And then it decodes that vector back into the data in such a way that you can reconstruct the data itself.

So the second half of the autoencoder, the decoder, it's doing functionally basically what we've described a generator, a generative model, as doing. So the second half of the autoencoder, the decoder, is a mapping from some z vector to data. And now the question is, can we use that decoder as a generative model, as a generator, to sample new random photos?

In order to do that, we need to be able to first sample a z vector and then decode it into an image. So the first question for you to think about-- what is the issue with this formulation? That's better. I don't know why that happened. So we have a mapping from z to data X. And to sample a new random photo from the decoder, we would first have to sample a new random z vector. So what's the difficulty here? How would I do that? Anyone can think about what would be difficult about sampling from some distribution over z. Yeah, let's go here.

**AUDIENCE:** What distribution?

**PHILLIP ISOLA:** What distribution? I never told you what z is. Yeah, in the back, I think I saw a hand. Right, how do you know what is the distribution over the encodings of an autoencoder? So that's the problem.

So typically, an autoencoder will learn a mapping to some distribution, some set of embeddings. But there's no guarantee that set of embeddings is going to form a simpler shape or easier to sample from distribution than the original data itself. So the problem is, I can decode z vectors that are corresponding to the data that I trained on. So if I have f of x, where x is a real data point, I can decode that by doing g of f of x. But I don't know how to sample new valid f's of x. It might be some weird distribution full of holes. And if I go to a point that has never been trained, it's never been observed via f of x, then I won't know how to decode that.

So the whole point of a variational autoencoder is to fix this problem is to make it so that the latent space, the encodings of an autoencoder, form a simple distribution that you can sample from. That's all of the math and algorithms that we'll develop in this lecture are going to be simply to do that, simply to make it so that this becomes a sample from distribution.

So we're going to go through VAEs from a few different perspectives. We're going to build up the kind of mechanistic intuition of what's going on. But we'll also look at the theoretical kind of probabilistic modeling way of motivating these things, and try to understand at least some of the layers of analysis that I'm going to provide. So it might be that you get lost in some of the math, but then always remember to pop back out.

So the first perspective I want you to understand is that a VAE is an autoencoder with one change, which is that rather than the encoder mapping to a complicated shape, it will map to a simple shape. It will map to a Gaussian. So a VAE is just an autoencoder that maps where f maps to a Gaussian, and that means that I can sample new images by sampling from a Gaussian and running the Gaussian random vectors through decoder. That's the first level of understanding of a VAE. Question?

**AUDIENCE:** Is the reason that we want the latent space to be Gaussian specifically because it makes it better to generate models? Or is this an intrinsic good [? of having the ?] representation space or something?

**PHILLIP ISOLA:** Yes, so why do we want the latent space to be Gaussian? So the level of understanding that I want you to have right now is that if the latent space were Gaussian, you could sample from it, and then it becomes a generative model sample. A Gaussian from z, decode, becomes an image.

Now you also asked, is it that a Gaussian is a better-- that leads to better representations? If I just want to think of this from a representation learning angle, is mapping to a Gaussian distribution a good idea? And that's also true, but it's a little bit more subtle. So we'll get to that toward the end.

So VAEs are great representation learners and great generative models, or they're good at both of those things. That's why we're getting to them after we've already covered basic representation learning and basic generative modeling because the VAE puts both of them together. And it encompasses all the models we've seen.

And many of the models we've seen, like diffusion models, can be cast as a certain kind of VAE or autoencoders or a degenerate VAE. So VAEs are kind of-- they're the pinnacle of generative modeling and representation learning. But in practice, they're not as popular these days as diffusion models and other things. But mathematically, they're a framework that encompasses almost everything that we've seen.

So what is the probability interpretation of a VAE? So now we're going to take a step back, and we're going to talk about mixtures of Gaussians because it turns out, variational autoencoders are fitting a mixture of Gaussians to a data distribution. So a mixture of Gaussians-- so what is a mixture of Gaussian? Well, we talked about-- we've seen Gaussians.

So a Gaussian fit to data with a max likelihood objective is just going to be a density model where we try to increase the density that we apply to where the data lives. So there'll be a gradient that pushes up the density here. And the parameters are just the mean. And the variance will shift around the mean and the variance of this Gaussian until we get maximum likelihood applied to the data points. And that's what we learned about last Thursday. So that's one Gaussian.

A mixture of Gaussians is just, well, in order to model a complicated distribution that does not have a Gaussian shape, I can use a bunch of Gaussian distributions and sum them up. So a Gaussian is like a little-- think of it like a little circle. I mean, it could be an ellipse, but most of this lecture will just assume isotropic Gaussians, which means that the covariance is 1 or is the identity, in fact. So we're going to assume, just for simplicity, that there's no ellipsoids or covariance structure. It's just circles. You can always turn these into ellipses, but in the pictures, I'll just draw circles.

So what does it look like to have a mixture of Gaussians that is fit to some distribution, like this funny shape that I was coloring in blue on the previous slides? It just looks like this. I'm going to say I can approximate that shape with a bunch of circles. If the circles are small enough and I have enough of them, then I can approximate any shape. So probably, you could prove-- I haven't thought through this, but probably a mixture of Gaussians can, to arbitrary precision, approximate any density, something like that.

So here's what the mathematical model is. It's just a mixture, so a sum, with weightings w. I think, in traditional models, these would be positive. But you can imagine these being negative or positive more generally. And every component is going to be a Gaussian, a little blip of Gaussian probability centered at a mean, mu, and covariance, sigma.

So the mean of the first mixture component is here and the covariance is-- in this case, it's just isotropic two dimensional Gaussian. So we're just going to have a covariance is equal to the variance. It's just one number that characterizes that shape. It's the radius of that circle. So that's the mixture of Gaussians. And the parameters of the mixture of Gaussians are the mixing weights, w, the means and the variances, the covariances of all of the Gaussians in the mixture. And in general, in high dimensions the covariance will be d by d dimensional matrix, where d is the dimensionality, and the mean will be a d-dimensional vector.

So VAEs are a mixture of Gaussians. But they're not just any mixture. They're actually an infinite mixture of Gaussians. So mixture of Gaussians is a classical model. That kind of new thing with a VAE is it's an infinite mixture of Gaussians. So how could I parameterize an infinite mixture of Gaussians?

So a mixture of Gaussians-- I have weight, so that's one parameter. I have mean. That's d parameters. And covariance-- that's d squared parameters. And I have k of those. So I have d squared times k plus times k plus parameters to specify k Gaussians. So to specify infinite Gaussians, I need infinite parameters. So clearly, that's not going to work, that explicit parameterization of infinite mixture components.

So there's a really cool trick for parameterizing an infinite mixture with a finite number of parameters. And that is to parameterize that infinite mixture as a mapping from some underlying base continuous distribution to mixture components via function g that has a finite set of parameters. And in this course, g will be a neural network. But in fact, a VAE doesn't have to have anything to do with neural networks. g could be any type of mapping. But in this course and in traditional literature, VAEs are always using neural nets. But yeah, really, you could apply this to other models, too.

So here's how it works. We have pz. This is a unit Gaussian distribution. We sample a random vector from that space. We pass it through our generator or decoder g. And that outputs a number, which we will interpret as the mean of one of my infinite mixture components. So I'll index the mean with this symbol mu, and we'll have the neural network g or the function g actually also output the variance. So it'll output a set of numbers, a d, a d-dimensional vector for the mean, and a d-squared-dimensional vector for the covariance or, in this example, two numbers for the mean and maybe one number to characterize the covariance because it's an isotropic Gaussian.

So if I sample different points in the z space, I will end up parameterizing different Gaussians. And because z is continuous, I can sample any point on a continuum, and I can thereby get an infinite number of possible or continuous possible set of Gaussian mixture components. And g just has weights and biases. It's a neural net. So that's a finite number of parameters to model an infinite mixture.

So this has parameters theta. And now we have fit some weird complicated distribution with an infinite mixture parameterized by a neural net with weights and biases theta.

So what is the density implied by this mapping? So it's going to be not a sum because we have a continuous space. So we have to switch to a continuous math and use integrals, so the probability of some data point x. So the distribution over x implied by this model is going to be an integral over all of my mixture components. Each mixture component is just going to be a normal distribution, a Gaussian, with mean given by the neural net and variance given by the neural net. That will place a little blip of density over x. And my total probability will just be marginalizing over z, so integrating over z.

So here's the finite set of Gaussian. It's the sum over my mixture components. But now I have an infinite set of mixture components. And I'm going to take the integral over all of those components. And that integral is over z space.

So this is the infinite mixture of Gaussian model. These are the weights. pz is my prior distribution. It's usually just a unit Gaussian. And think of that the weights of the mixture components. And then these are all the mixture components. And it's an integral. Any questions about that? So this is the probability model that underlies VAEs for a Gaussian VAE. You can actually have non-Gaussian. So you could have other mixtures. It doesn't have to be a mixture of Gaussians. It could be a mixture of Laplace distributions or Poisson distributions, whatever you want. So it doesn't have to be Gaussians. But usually, we'll work with Gaussians.

**AUDIENCE:** Is this a convolution, or is that the wrong way to look at it?

**PHILLIP ISOLA:** Is it convolution? I suppose it could be interpreted as convolution. Like you're convolving over z-space with-- yeah, I'm not sure that's the most useful way of looking at it, but you could probably cast it as convolution in some sense. Yeah.

OK, so I think it's easier to think of it as marginal likelihood and marginalizing over some latent variables, so that's what we'll see on this slide.

So we're going to understand the z-variable-- the z-space as being latent variables that describe all the attributes of the generated data, but in an abstract fashion, and all those latent variables, we'll hope that they have a simple distribution.

And they'll be unit Gaussians in the VAE, but in the example I gave last class, I said that I could write a Python program that will take noise from simple distributions and generate images, and these simple distributions are my distribution over the latent variables, and they have interpretations.

So the graphical model-- if you've done probability with graphical models it would look like this. We sample a z, and then condition on the z. We sample an x, we only observe the x. The latent variables are not observed. Our data modeling problem is to fit a distribution over x that is represented as a mapping from some underlying variable z. But we don't observe z. We have to maybe infer z from x.

OK. So the generator will be-- think of that as a conditional distribution, the probability over x given some setting of the latent variables. And if I want to model the likelihood of a data point x, I can integrate over my latent variables z. So this is just marginalization from probability. Px given z times pz is px and z. Then if I integrate over z, then by the rules of probability, I get px.

OK. So this is called-- this is-- the likelihood the probability that the model p theta places on the data is called the likelihood, and I'm trying to do max likelihood learning. So I want to find the probability model p theta that maximizes the likelihood of the data that assigns the most density to the data.

And to compute the likelihood, I have to marginalize over my unobserved latent variables. So sometimes we call this marginal likelihood. So a latent variable model, I'm going to fit it to data by maximizing the marginal likelihood that it assigns to data when I marginalize over all the latent variables.

Because for a given data point, there might-- to make it high probability, there might be a lot of different settings of the latent variable that can give it some plausible-- like, a high probability. OK.

OK, so here is the entire objective function and hypothesis space of a variational autoencoder, or in this case, just an infinite Gaussian mixture model, and the variational autoencoder can be-- is one type of approach that uses this model.

So the objective function is going to be to do maximum likelihood. So I'm going to maximize the likelihood that my model places on the data. And the hypothesis space is going to be an infinite mixture of Gaussians. So I'm going to try to find the means and the variances and the weights of Gaussians in an infinite mixture, and to parameterize an infinite mixture, I'm going to parameterize a mapping from some underlying prior space z to the data space x via the math that we just looked at on the last few slides.

OK. So this is marginal likelihood. I'm marginalizing over the latent variables z. And the likelihood term for px given z is going to be Gaussian, and the prior term for pz is going to be unit Gaussian, mapping from unit Gaussian to complicated data distribution in that fashion.

And this model is going to give me two things. It's going to give me both a density-- that's the main thing. I'm trying to learn the parameters theta that will define this density function that maps from x to non-negative numbers. And it will actually also give me a very easy way of sampling from that density. So it gives me those two things.

So in order to sample from this density, I can do the following. I first sample a random unit Gaussian. So that's sampling my z vector, And then I decode it through g.

OK. So in order to create a sample x, I first sample a random unit Gaussian z. Or, in fact, this sampler that's written here is just going to be the mapping p of x given z. So it's going to say given some z, how do I draw a sample over the possible x's that follow the density px given z?

So I will take my z, and I will have my neural network predict what is the mean of the mixture component, and then I will transform-- I will take a sample with some variance by multiplying my variance by a unit Gaussian.

So this is just a way to sample from a Gaussian whose mean is g of mu and whose variance is g sigma by transforming a sample from a unit Gaussian. So x will be distributed as normal distribution whose mean is g of mu and whose variance is g of sigma.

OK. So the difficulty here-- let's see. Sorry. The variance is sigma squared. I suppose sigma is the standard deviation, not the variance. I always mix up those two. OK. So, yeah, the variance-- did I get this right? Variance is square of standard deviation. OK, yeah.

So the difficulty here is this term right here. So if I want to maximize the likelihood that my model assigns to the data, well, the likelihood is written as an integral. So if I were to try to solve this with back-propagation or something like that, that means that every time I'm going to perturb my parameters to place more probability, more likelihood on the data, I would have to compute this integral. And this would be incredibly expensive.

So we don't even have a closed form for that integral, so we would compute that integral by approximating it with a set of samples by approximating the integral somehow. To get a good approximation to an integral, I need to maybe take a lot of samples over z.

And this means that every time I'm going to run my model forward, see what density it places in my data, I actually have to run it forward a lot, a lot of times because look at the neural network, and g appears inside the integral, so to get a good approximation to the likelihood my model places on the data, I have to run my neural network forward as many times as necessary to approximate this integral, which might be a lot of times.

So one step of backprop is running my neural network forward many, many, many times to get a good estimate of this integral, and then I still have to backprop that.

OK. So this integral is difficult to calculate. It's an intractable integral, so what am I going to do? Oh. And just notation-wise, there's something that you might encounter in the literature, which is called the reparameterization trick, which just refers to this. In order to take a sample from this distribution, px given z, which is a normal distribution, it suffices to take a sample from a unit Gaussian and transform it by offsetting it by the mean and then multiplying it by the variance. OK.

So then x is just a deterministic function of g with the only stochasticity coming in through this unit Gaussian noise inserted in the side. It's just a little trick.

**AUDIENCE:** Just to clarify. So you say a variance because only sigma squared, is it standard deviation?

**PHILLIP ISOLA:** Yeah. Sorry, I got mixed up in the terminology. So sigma is the standard deviation, sigma squared is the variance. And, yeah, apologies for mixing up those terms.

**AUDIENCE:** So is this-- it should be standard deviation or is this a typo, it should be sigma squared?

**PHILLIP ISOLA:** No, I believe that it should be multiplied by the standard deviation, yeah. I think this is not a typo, it's just I misspoke. OK. Anybody can confirm that? That's the proper way to transform a unit Gaussian into a Gaussian with some variance? OK, good. So you multiply by the standard deviation. Square root of the variance. OK.

OK. So that's the problem that we're trying to solve, and the rest of the VAE trickery is going to be, how do you calculate this integral in an efficient manner? So you could pause here and you could say you understand what VAEs are doing.

They're modeling the-- they're maximizing the data likelihood by fitting an infinite mixture of Gaussians to the data and doing optimization to increase the probability we place on x with respect to the parameters theta, which are the weights and biases of the neural network g. That's one layer of understanding. But then there's this integral-- so the rest of the lecture is actually how do we approximate that integral. OK.

OK, so I'm going to use the notation L to represent the objective function, which is the data log likelihood. So L is equal to this equation down here. So that equation is the marginal likelihood-- so it's going to be the probability of x given z times z integrated over z is px. Log of px is the log likelihood. I'm trying to maximize the log likelihood my model places on the data.

And this outer sum over n data points is going to be-- well, my training set. I'm going to be fitting my model to a training set of n observations. So maybe I have n photographs and I want to learn a generative model of them.

So I-- since this is log likelihood, when I have multiple different data points, I maximize the joint probability by maximizing the sum over the log probabilities. OK, so this is the objective function for maximizing the likelihood using a mixture of Gaussians.

OK. So I'm now going to introduce a series of tricks-- I think around three tricks for approximating this integral. So the first trick is going to be the most vanilla trick that we use all the time, which is to approximate an integral I can use some sample-- I can use sampling in some form. And the nice thing about this integral is it's actually in the form of an expectation.

So this marginal likelihood, marginalizing over the latent variables, is equal to the expectation of how much likelihood probability density I place on x for each z in expectation over random samples from the prior pz. So that's the definition of expectation. It's an integral with-- if I sample from this distribution, it goes outside the expectation, this is just a definition.

OK. So, how do we approximate expectations? You can approximate expectations by simply taking a finite set of samples and averaging them. That's the trick that comes up all the time in math. If I have an expectation, I approximate it with a Monte Carlo estimate, which is a set of samples, and then I take the average, as opposed to having to compute the entire integral.

As M goes to infinity, this becomes-- this is an unbiased estimator of that integral. So as M goes to infinity, then I get a perfect estimate.

OK, so how do I do that? I sample from my prior, which is unit Gaussian. That's easy. We can sample unit Gaussian vectors. And then I sum up and take the average of the density that the model places over x for each z. And remember that px given z is a very simple distribution. That's just a Gaussian distribution. That was here. So px given z is just a Gaussian distribution parametrized by my neural net output mean and variance. OK.

So this is trick number one. And now that we have done that trick, we actually can train this. I can actually write PyTorch and do this because I got rid of the integral, I don't know how to write integrals in PyTorch. But now I've got it into a form where we can write it in PyTorch. Everything is just countable, discrete math.

So I approximated the integral with an average over average over M samples from my prior, and I looked at how much probability each of those samples places on my data. I add it all up. And then I will-- it's a log probability, so I'm trying to estimate log of p theta x. So log of-- this is p theta x. This is my approximation to p theta x.

And then I will be-- my total probability over all my training data points is a sum over the n data points, and I'm going to try to find the parameters that maximize the probability I place over all of my data points. OK. Ah. I see-- I think I-- no. Oh yeah, this is fine. OK, good.

Yeah no, I think actually, I think there's a typo here. So that 1 over M should be inside the log, I suppose. Yeah, that's probably an important typo. OK. So log 1 over M.

So there's a lot of equations in this lecture, so be sure to try to go through and think about them yourself, and there might be typos. So exercise for you to solve.

OK, so what does that look like? So here is actually a colab that's running this optimization to fit a mixture of Gaussians to this data distribution on the right. And rather than showing random samples-- so rather than randomly sampling from the prior pz, I'm sampling along a grid of points just for visualization.

So we can say, for this z, that maps to a Gaussian at this location. Now at initialization, which is the frame that you're currently seeing, I've initialized the neural networks so the weights are near 0, and so the Gaussians-- all the mixture components are mapping to around 0.

OK, so here's what it looks like when you train this system. So the different points in z-space learn to map to different parts of the data distribution, so they cover the data distribution, they maximize this likelihood. OK, so that's what it looks like. This is SGD on the equation above OK. Yeah, question?

**AUDIENCE:** Why don't we multiply by the probability of z?

**PHILLIP ISOLA:** Why don't we multiply by it? Yeah, great question. So that's just because of how expectation works. So expectation is defined as this integral, and the probability z comes in through the sampling procedure. So higher probability samples, according to z, will be overrepresented in this sum, so don't need to double-weight them. Yeah, that's why.

**AUDIENCE:** I was just a little confused on how are you computing p theta of x given z. You said it was just-- I mean, like, on the previous slide, it's like a normal distribution with a mean and standard deviation or variance provided by the forward pass through your model. So then do you just draw a sample from the normal distribution defined by the parameters that your neural net--

**PHILLIP ISOLA:** You don't-- so you don't have to draw a sample because this thing has a closed-form analytical form. So what is n of x with mu and sigma as your-- or sigma squared, let's say, as your mean and your variance? This is equal to some normalization times e to the negative x minus mu over 2 sigma squared. That's just the analytical form of the density, yeah. So this thing here is just shorthand for some exponential of the difference from the mean divided by the variance.

**AUDIENCE:** In the approximation step, how does the error scale with M?

**PHILLIP ISOLA:** In the approximation step, how does the error scale with M? I don't know. That's a great question. So I think that it's going to depend on, yeah, the data and the parametric form of p and all these different things, but I would just say, that's like a good question to do some analysis on, and I'm sure people have-- in general, how do Monte Carlo estimators scale with respect to the number of samples? And there's probably some classical results, I don't remember. Yeah. OK. Yeah?

OK. So, that's all all well and good. So actually, I'm going to ask a question related to that-- how many samples do you need to get a good approximation? So I'll come to your question in a second. So how many samples do you need to get a good approximation?

So this worked. Do we need any more tricks? Is this enough? So here, I took some number of samples-- let's say M is like 100-- I don't remember the exact number. Well, what's going to happen as I scale my latent dimensionality? And I'm going to need a high-dimensional latent space in order to be an effective model of a high-dimensional data space.

So for two-dimensional distributions like shown here, it turns out that I can just sample 100 of these samples M and get a good fit to my data distribution. But for high-dimensional data, I need a high-dimensional latent space-- or higher-dimensional latent space.

So let's say that my z vectors are hundreds dimensional. Do you think 100 samples M are going to be a good approximation to the integral over this 100-dimensional space? What do you think? Thumbs up if you think it will be OK? OK, no. Thumbs down if you think it won't be OK? Yeah, thumbs down.

OK, so why? How many samples, roughly, would you say you need to have a good approximation to an integral over a D-dimensional space? This kind of gets at your question, I think. Yeah, in the back.

**AUDIENCE:** Responding to his question, your question, the Monte Carlo estimator has variance 1 over M, so it goes down to square root-- 1 over square root of M.

**PHILLIP ISOLA:** Excellent. OK, thank you. Monte Carlo estimator has variance 1 over M.

**AUDIENCE:** Nominally non-dimensional, but in practice, somewhat dimensional.

**PHILLIP ISOLA:** Nominally not dimensional, so actually, there's no dependence on the dimensionality D, I guess, but in practice, there is. OK.

So I'll say that my rough intuition would be that you need exponential samples. In order to cover a D-dimensional space-- and each dimension is going to be-- you're going to have to fill it up with another fixed set of samples, so you should have something exponentially more samples to cover a D-dimensional space. But yeah, formally I'm not sure what-- so actually, how does that work? I'm interested. Yeah. Central limit theorem?

**AUDIENCE:** Yeah. The classical result.

**PHILLIP ISOLA:** Yeah. OK. OK, well--

[INTERPOSING VOICES]

**AUDIENCE:** --multiple dimensions [INAUDIBLE]

**AUDIENCE:** Yeah, [? it can happen. ?] We end up with a multi-dimensional Gaussian as your limit, and that has a multi-dimensional variance. And then if you want to calculate the error of that instead of variance, I think exponential, as I mentioned, is exactly correct.

**PHILLIP ISOLA:** I think it's exponential in dimension-- so there should be some term in terms of-- the number of samples you need to get some quality of fit should depend on the dimension. I think it's exponential in dimension, but there's probably more to discuss, maybe we can take that offline.

But I'll just say that in practice, this doesn't work in high dimensions, and I think it's because you need exponentially more samples to get a good approximation to the integral in high dimensions, but we should talk about that more offline to see if that's really true.

OK. So, the next trick of the VAE is we're not going to just sample randomly, we're going to sample intelligently. We're not going to just do naive Monte Carlo approximation to the integral, we're going to do a much more intelligent approximation to the integral, and that one is going to be called importance sampling.

OK. So what is the idea of importance sampling? The idea is that let's say that I want to estimate the likelihood that my model places on some data point x, that little x up there. OK, what I'm going to do with importance sampling is say, I could say that that's the expectation of px given z with respect-- taking expectation over z. And I could randomly sample the z's to calculate that expectation.

But what will happen if I randomly sample the z's? Then almost every single px given z for most of the z's will place nearly zero probability on that data point x. Most of these mixture components don't lie near the data point x. Just in practice, this is what might happen if I've spread out these mixture components to be a good fit to the data density.

OK, so the observation is that with random sampling, maybe I have to take around 7 samples to get one Gaussian component that's near x, and almost all the other components are adding 0-- roughly 0, so it's like this. p of x is equal to the average of px given one sample plus px given the next sample plus px given the next sample, but most of these px given a sample are 0.

So almost all the terms are nearly 0 except for the sample that-- except for the z's that map to Gaussians that place high density over this x. So in this expectation, most of the terms are going to be 0 for any given point x.

So we can exploit that by just trying to say, let me only take the sample z2, and that will be a good approximation to this expectation. And this is called-- this is the idea behind importance sampling.

So in order to approximate this expectation, which is that integral, I can simply multiply by q-- by another density q, and I can multiply by qz divided by qz-- so qz divided by qz will be 1 as long as qz is non-zero everywhere, which is just a technical condition, we need to make sure it has support everywhere.

OK, so then I can-- that's valid. So I'm just multiplying by 1. But the interesting thing here is now this appears as an expectation with respect to z samples from qz as opposed to z samples from pz.

And if qz is a distribution which has placed high density only on the z's that actually contribute to the sum that actually have high probability of the data under that z, high px given z, then this will be an expectation which requires fewer samples to get a good estimate of the true integral. OK, so this is called importance sampling. I'm sampling from a distribution that gives more importance to the points that matter.

OK, so I won't prove this, but I'll just assert that the optimal distribution to sample from for performing this estimation. In a certain sense, I think the distribution that minimizes the variance over my estimator of this expectation, is to sample z's from exactly the density p Z given x. OK.

So what does that look like? So it's like if I knew p Z given x, it's like, here's x. So given x, these are the Z's, let's say, that place high probability on x. It's a distribution-- it might be a complicated distribution, but most of the density-- like, this is the high-density region of p Z given x, is in a small region.

And if I could simply sample from those points, it would suffice to get a good estimate of what would happen if I had sampled from everywhere because most other points will place zero density and they just won't contribute to that integral-- or roughly zero density. OK.

OK, so importance sampling is like now I just sample once from this distribution and I get as good an estimate as having sampled seven times with random sampling.

OK, so here's the entire transformation. I'm going to estimate the likelihood that my model places on the data via this expectation, but I'm going to switch it to this other expectation where-- these two expectations are exactly equal to each other. That's because these two integrals are exactly equal to each other if q has non-zero support everywhere.

But this one can be approximated with a Monte Carlo set of-- a Monte Carlo approximation to this-- expectation is going to be a better estimate, lower variance than this expectation. OK.

So here's an importance sampling. Random sampling, I take random samples from z-- from pz. And most of them don't contribute to the sum. Importance sampling, I only sample from the points that contribute to the sum. I only sample from the points that the model thinks will place high probability on x. So the probability of z given x is the points that the model thinks are the z's that will place high probability on x. OK. Great.

So, if I knew p Z given x, then this would be what I want to do. I just sample from p Z given x and I get a better approximation to that integral, and this will then work in high dimensions because for high dimensions, the volume is exponential in D, but the area of z-space that actually maps to any given point x might be very small under a particular model p.

OK. But the problem is, I don't know p Z given x that might be another complicated distribution, and I now have to come up with a way of modeling that distribution. So here's how you do it.

The next step in the VAE-- the third trick is you're going to try to model or predict the optimal sampling distribution, the sampling distribution p Z given x given x. You're going to train a model that will take x as input and output a density over Z, and that will be a model of p Z given x.

OK, so now we get to where it starts to look like an autoencoder. It probably didn't look at all like an autoencoder until now, but what does that look like? That's an encoder. That's a mapping from x to a distribution over z-space.

So our model of-- our model, which we'll call q, which is trying to approximate p theta Z given x, so we're going to try to approximate that with another prediction called q. That density is going to be, just for simplicity, modeled also as a Gaussian.

So we're going to take our x, we're going to try to predict what part of z-space would p place high probability on x. So for what part of z-space is px given z high probability? We're going to predict that by modeling a distribution over z-space.

And we're going to make that a Gaussian. It will be-- we'll take our neural network f, it will output a mean and a variance, or maybe a covariance if we wish, and that will be our model for pz given x, and that model is called q. And q has parameters psi, which is a new set of weights and biases of this encoder function. OK.

OK. So, now we're going to get to the full picture. So we have these two pieces. We have the decoder function, which is this probability model-- probability of x given z, and we have this encoder function, probability of z given x. We're going to put them together now.

So how am I going to actually write down the objective function for the encoder? So the objective, we're going to say we want it-- we want the encoder to model a distribution pz given x, and it's going to model that distribution with another distribution parameterized by mean and variance that defines a Gaussian, and that's going to be qz given x.

And we're going to try to make those two distributions the same. So we're going to use the KL divergence to try tc minimize the difference between those two distributions. Just a particular divergence measure of how different our two distributions, we want to make them the same.

OK, so this is our objective for the problem of the encoder. The problem of encoder is to minimize Jq. Actually, I think I have a negative sign, so it's going to be to maximize Jq.

So I'm just going to rewrite the KL divergence. There's a little bit of algebra that you'll probably want to look at in more detail on your own time, but this is one way of writing out the definition of KL divergence. It's going to be, yeah, this equation. It's going to be in-expectation sampling from the density that q Z given x all of these terms. OK.

So the learning problem for q is to maximize-- to minimize the KL divergence or maximize the negative KL divergence. So this is the learning problem for q. It's trying to predict the distribution over latent variables that will explain a given data point x. We call that inference sometimes, infer the latent variables that explain the data x.

I can take this equation and I can notice that if I'm maximizing with respect to the parameters of q, well, the parameters of q don't appear in this term, so I'm going to just remove that term. They don't affect the maximization problem, they don't affect the arg max over the parameters psi, the parameters of the encoder. So now-- and we'll use this term a little bit later, but this equation here will be J, it will come back in a minute.

OK, so for our learning problem for q is to optimize this. And this is an expectation, so I could optimize that again by just taking a Monte Carlo estimate to the expectation by sampling a bunch of different z's and taking the average over all of the values within the expectation. OK.

So that's the learning problem for q, which is parameterized by a neural network f. Map from x to an inferred distribution over latent variables that can explain x.

OK, the learning problem for p-- this was our original learning problem. We wanted to learn a generative model parameterized by a neural net g where g places Gaussian probability over the data space x, and g defines an infinite mixture of Gaussians by mapping from-- by taking an integral over all possible points in z-space.

OK, so for a learning problem for p, what I'm going to call Jp, is to find the parameters theta that maximize the likelihood that my model places on the data, and with importance sampling, that means I'm going to sample from my importance sampling distribution q, which we learned the optimal importance sampling distribution q up here, we know that it should be pz given x, but we're just going to try to learn z given x by fitting a distribution q to pz given x.

And this is, then, my importance sampling for estimating the integral that defines the likelihood of the data under the parameters theta of my model neural net g. OK.

So I know there's a lot of terms in math going on. This might take you time to work through step by step. All of this derivation is in the reading for today, so these slides follow that almost exactly. And additionally, in the problem set, you'll get to work through some of this yourself, so hopefully you'll enjoy that. But I know this is a little more mathematically notation-heavy than some of the other lectures.

OK. So now putting everything together, there's one little final step, which is, it turns out that if we approximate log of expectation by taking sample like a Monte Carlo estimate-- by taking samples from the thing that we're taking the expectation over, if we do a Monte Carlo estimate of this and then we take log of that, it turns out, that's a biased estimator of log of expectation.

So Monte Carlo estimate of log of expectation is a biased estimator, and we don't really want that. We want it to be the case that as samples go to infinity, we get the true answer, and that won't be the case here.

So we're going to do this little trick, which is that we're going to pull the log inside the expectation so that then we just have expectation of some function, and then a Monte Carlo estimate of that expectation is an unbiased estimator of that expectation.

OK. So we're allowed to do that by Jensen's inequality that says log of e of some function is equal-- is greater than or equal to expectation of log of that function. So this is just an identity, again, from-- a lot of you will have seen Jensen's inequality. It comes up a lot when you're working with probabilities.

And now what we have is that this is going to be a lower bound to our objective J. So this is a lower bound. I can rewrite it-- so I take the log, I move around the terms a little bit. This is just rewriting some algebra. And this thing here is called the evidence lower bound. It's called the ELBO. So you'll come across this term if you're working in VAEs or other types of what we call variational inference.

And it's a lower bound to the thing we actually care about, which is the likelihood of the data under the model. And it's a lower bound due to Jensen's inequality on this line. And this is something that is now-- we can approximate this expectation as an-- or this expectation with samples and it'll be an unbiased estimator of that value.

So we're going to have an unbiased estimator of the lower bound, we can maximize the lower bound on the likelihood the model applies to the data, and this will learn a model that places high likelihood on the data. Maximizing the lower bound will increase my objective J. OK.

So evidence-- like, this is the evidence that the model provides for the data. Anyway, yeah, don't worry about the name too much, but it's called the evidence lower-bound, it's just terminology, ELBO. OK.

So it comes in two terms that both have intuitive form, and I'll get to those on the next slide. But the interesting thing is, once we have written things in this form, what you can notice, if you go back a slide, is that this objective is actually exactly equal to the objective that we had for q.

So q appears over here in this KL term. So what is this saying? This is saying that in order to learn the parameters of the encoder that give me the best importance sampling distribution for estimating the likelihood the decoder places on the data, it actually works out that I can just maximize this evidence lower bound with respect to the encoder and decoder parameters jointly.

OK. So, the VAE objective now can be written in this form as maximizing this lower bound to the thing we actually care about, and I can approximate that with sampling, and it will be an unbiased estimator of all of these expectations and integrals.

It has two terms. It has the first term, which is how much probability does the model place on the data? And another term, which is, well, I was estimating how much probability the model places on the data with respect to the wrong-- not actually taking samples over all of z-space, but with taking samples with respect to this other distribution. So I have to somehow account for that, so I'll penalize the difference between this other distribution in my prior, the unit Gaussian pz.

OK. So this term's coming out because I had to multiply by this ratio. How much different are my samples under pz from my samples under qz? OK, so that pops out into this term here. So now, the encoder is trying to make these things as similar as possible, and the decoder is trying to just learn parameters theta that will maximize the probability placed on the data.

OK. So here's the picture. I'm going to start with a data point x. I will encode it. That will define a distribution over my latent space. I'll take a sample from that distribution, that's one important sample, for estimating the probability of my model places on the data. And then I'll decode that and see what probability the model did place on the data. That's one important sampling step of approximating the lower bound to the probability that my model places on the data. OK.

So, now we come back to autoencoder. A lot of math, but at the end of the day, the VAE is a very simple model that looks just like an autoencoder. So here's how it looks like an autoencoder. We start with data point x, we encode it, we get mean mu z and variance sigma squared z.

We then take a sample from the distribution q Z given x. So taking a sample, that means we're just going to add noise. So remember that mu z plus unit Gaussian epsilon times standard deviation for the encoded point x-- so sigma z is going to be the standard deviation of the encoded point x.

This is one sample from a normal distribution centered at mu z with variance sigma z squared. OK, I think I'm getting the squared versus not squared term correct now.

So, that is one sample, for importance sampling, for estimating the density that my model will place on the data. So then I will decode that to observe, under a normal distribution, what is the probability that my model placed on that data for that mixture component, which is a good approximation to the entire integral.

OK, so remember the normal distribution is just an analytical form, so I can calculate how much probability g placed on the data point x.

OK, so this looks just like an autoencoder except one step, which is this noise I added in the latent space. Otherwise, it's just encode-decode, but I added a little bit of noise to the latent space because that's my sampling. That's necessary because I'm sampling to approximate this expectation.

OK, so here's the two terms of the ELBO that I talked about before. The first term, it looks like a reconstruction error. Why? Because p theta x given z is going to be a normal distribution over x centered at a mean, which is the decoded g theta of z. So a normal distribution over x is just going to be-- a log normal distribution will be just the squared error. If I take log of e to the negative squared distance, it's just negative squared distance.

So how different is my decoded point from my input observation x? That's this likelihood term. And these other terms over here are just-- because the Gaussian distribution has to be normalized, and the variance appears in the normalization, and dividing by 2 sigma squared is just also part of the definition of the Gaussian probability.

So just for simplicity, imagine that we're going to model-- we're going to use a VAE where we won't actually model the covariance or even the variance of the Gaussians. This is a valid modeling choice. So I could just fix my circles to have radius 1. I can say I'm going to try to fit an infinite mixture of Gaussians where I don't get to change the variance of the circles, but I'm still going to try to find the max likelihood fit. That's a perfectly valid modeling choice.

And if I did that, then sigma becomes 1, and these terms all become constant. And then you can see that really, all I would be doing is trying to minimize the difference between my data point x and encode-decode of the data point x. So then it would be exactly the autoencoder objective of minimizing the mean squared error between my reconstructions and my input observation.

So VAEs often will also model the variance of this Gaussian, but vanilla autoencoders don't really model the variance of the Gaussian, they just kind of implicitly set it to be constant. OK.

OK, the other term, the KL, is saying, well, I need my q Z given x distribution. I want it to be close to a unit Gaussian. So we say that the posterior of the inference direction should be near the prior pz So what is this? This term also has a very simple interpretation. So here's what it looks like. This is the KL-- this is the KL divergence between two Gaussians.

So luckily, the KL divergence between two Gaussians has a closed form, an analytical form, and it looks like that. So remember, this is the KL divergence between the Gaussian defined by q Z given x and the prior Gaussian, which is a unit Gaussian. So this is the KL divergence with respect to-- between a Gaussian that has mean mu and variance sigma squared, and a unit Gaussian that has mean 0 and variance 1.

OK. It has this form. In the readings, we posted the original VAE paper, and in the appendix, there's a nice derivation of that form. OK, so what does this look like? Well, let's, again, imagine that we're not going to model the variance, we're just going to set the variance to be one constant. We're only going to get to update our neural net to improve the density by changing the means that it places over the data.

OK, so these terms would then can be ignored, they go away, this is just a constant. So what is that? That's going to be I want to minimize the means. So basically, if all these terms are constant, one half is just a scaling, in order to increase the evidence lower bound, in order to place higher probability on the data, increase the lower bound to the likelihood I place on the data, I'm going to try to decrease the means.

So all of this term is saying is squash the means toward the origin. So I'm going to try to reconstruct my data-- that's the first term, that's the data reconstruction term, that's an autoencoder. And I have one extra term, which is just do that in a way where all of the encodings are near the origin.

So what does that look like? That means that this funny shape that I could have gotten, this is an autoencoder-- I just encode, I decode, I minimize reconstruction, I might get an arbitrarily complicated latent space. In fact, here, if my latent space is two-dimensional on this cartoon, I could just have f be the identity, and then my latent space has exactly the same shape as my input data space.

If g and f are both the identity, it's just, like, I encode to the same shape, I decode back to the same shape. That would satisfy the reconstruction error, but it wouldn't be a very good representation of the data, it would just be the raw data itself once again.

So the variational autoencoder simply says, no, you've got to squish that latent space toward the origin. So this means that in a normal autoencoder, I need to make sure the dimensionality of the latent space is low so that I'm doing compression and forcing it to do something non-trivial.

In a variational autoencoder, it helps to have a low-dimensional latent space, but it's less critical because there's this other term, this KL term, which is trying to compress the distribution toward 0-- compress all the means towards 0. And equivalently, we can understand that to be, I'm trying to make the distribution of encodings z have the shape of a unit Gaussian, but when I'm not modeling the variance that just means squishing the means towards 0. OK.

So I just squish everything down. Why does it not collapse to a single point? So what do you think? If I'm trying to squish the means towards 0, why don't they just all collapse to 0? What do you think? What prevents that? Yeah? I think I heard an answer. No?

**AUDIENCE:** Would it push everything to 0?

**PHILLIP ISOLA:** If you push everything to 0--

**AUDIENCE:** That's trivial.

**PHILLIP ISOLA:** It's trivial. And what will happen to the reconstruction?

**AUDIENCE:** You cannot.

**PHILLIP ISOLA:** You cannot. Yeah. So that's the tension. We're going to try to push everything to 0, that's this term if I'm not modeling the variance. But I also need to satisfy this term, which says that you better be able to reconstruct your input point, and if everything goes to 0, then there's ambiguity. So in order to reconstruct the input, I need to to preserve information. I can't compress too much-- I can't squish everything down too much, so you want to spread out in order to be able to reconstruct the data.

And that's the fundamental tension of the variational autoencoder. Pushes everything toward the 0, creates this compact shape. It's like I have some Play-Doh, and I'm trying to squish it down, but there's also this physical force which is preventing it from going to a black hole. It's like, no, no, I don't want to do that because I need to still be able to know where I came from to reconstruct the data.

It's like that. I'm pressing it down, but there's some tension. Eventually it gets to this unit shape, and that's what the math works out to. So here's what that looks like.

So now here, I'm showing the encoder mapping at initialization. The neural net is initialized with weights and biases that are near 0, so everything maps to 0. And then the decoder. So initially, all of my mixture components are near 0.

And here's what it looks like to train. So what you'll notice is I'm giving-- I'm getting a good coverage over the data distribution. That's my infinite mixture of Gaussians, but I'm visualizing it with a finite set of samples along this grid.

And what is the encoder doing? It's like, I want to spread out these Gaussians so that if I go from x to z and reconstruct, I'll know where I came from, I'll have enough information, but I also want to be near the unit sphere, unit Gaussian. So this tension will make it pop out and then squish back together.

And you can think of this almost like some sphere-packing problem. There's probably some interesting connection to a geometric way of understanding this, that if you have a bunch of spheres and you want to fit them all into-- you want to make them not overlap. If they overlap, what's the problem? That means there's ambiguity. If two different data points map to the same point, or overlapping Gaussians, then there's ambiguity, you don't know which one it came from, and you can't reconstruct.

So it's like, you don't want them to overlap too much, but you want them to squish down at the same time and there's this tension. And then the solution is that they'll spread out to evenly tile the unit Gaussian.

OK. So here's what the densities look like if I plot them as a heat map. So remember, these are the Gaussian components, but actually, they're continuous. They're little blips of Gaussian probability. So here's what it looks like as a heat map, it really did learn a good density model. And here's the density over the latent space, it becomes kind of a unit Gaussian in two dimensions. That looks like just-- like a little blip of bell-shaped probability.

OK, so VAE is really just three tricks-- approximate an infinite mixture with samples, sample efficiently by not just sampling at random, but with importance samples-- so approximate with important samples, and predict the optimal important samples to take the optimal importance sampling distribution with another neural network, and then jointly optimize the whole thing and it looks like an autoencoder.

OK. So in case some of that was a little bit too abstract, I just wanted to give you the step by step, like, what would you write in PyTorch to make this happen? It's actually very simple. So to optimize a VAE, I'll sample a data point x from my training set-- or maybe a batch of data points. I'll encode the data with a forward pass through my encoder.

For each data point that's been encoded, I'll add some noise to my encoding. That's my important sample. Then I'll decode the data by passing it through the decoder. It'll compute the losses, which will just be the squared error and the KL term. And then that will give me a scalar loss value at the end of the day, and I'll just backprop through that computation graph.

And typically for the VAE, even though I would really want to take a lot of important samples to get a better estimate, usually people just take one. They do forward one sample, decode backward, and that seems to be sufficient. Yeah?

**AUDIENCE:** How do you backprop through sampling?

**PHILLIP ISOLA:** So that's why that reparametrize. How do you backprop through sampling? That's why that reparameterization trick was important, because the stochastic element comes in through the side of the graph and you don't actually have to backprop through that sampling procedure. Yeah.

You're just backdropping through the mean and the variance of a distribution, and the actual noise is coming through the side and doesn't affect the optimization. A question in the back.

**AUDIENCE:** In practice, is it important to get the variance inside the latent, like sigmas, from the neural network? Or does it also work to just have always the same one? Because it would be natural to think like, oh, we should just do an autoencoder and you just noise it. Some things will be smoother than the representation, but maybe it doesn't matter to have different variances for different--

**PHILLIP ISOLA:** Yeah. So great, yeah, good question. The question is, for the encoder, is it important that the noise you're adding actually is dependent on the variance predicted by the neural net?

So you're kind of asking, for q Z given x, which you're modeling, would it suffice to just model the mean and the variance to be 1 or some constant? And I think, indeed, oftentimes you can get away with those types of things. Like, you can just take an autoencoder, add a little noise to the latent space and do nothing else, and it might be fine. Or maybe you add this KL penalty to say, take an autoencoder, penalize the means for being far from the origin, and add a little noise, and it'll give you a VAE and it might be fine.

But your mileage will vary, and you can always also not model these things with Gaussians, but more complicated distributions. So there's a lot of variations on this, but the simplest vanilla version might be good enough. And I think it captures the full idea.

Like, the simplest vanilla version is just smooth out the latent space and make it dense by penalizing the encodings from being far from the origin, and smooth it out by adding some noise so that it has finite information capacity, it can't overfit or have too complicated a decoding mapping. OK. Anyway, more to say about that. Yeah, question?

**AUDIENCE:** Is there a reason why pushing the latent variable distribution to 0 is better other than mathematics?

**PHILLIP ISOLA:** Yeah. So, right. So why is pushing the latent variable distribution to 0 better. So it comes out as that is what KL divergence between a unit Gaussian and another Gaussian will do.

But I think more intuitively, it's just-- it's this kind of sphere-packing an idea that I want it to be the case that every single point in z-space, if I sample from that point, it will lead to a valid image or a valid decoded data point x. And if I don't pack things in, if I don't squish all the means towards 0, then there might be gaps.

So, like here, there might be these gaps. At the end of the optimization, there's a little gap here. That might-- I mean, this is only a finite set of samples, but maybe there's no-- these data points here, like, they can't be reconstructed at all.

Or equivalently on this direction, if there's gaps here, if these means all went out to infinity-- so it's like, oh, I have high capacity, I can memorize everything, they're going to go out to infinity, so they'll all be infinitely far apart, and then I can more easily decode where they came from-- there's no overlap at all, it's like perfectly information-preserving, it's invertible, well, that wouldn't be good because there'd be all these gaps.

Where if I now take a sample from my z-space, I might end up in a gap. That's never-- so the decoder has never seen that sample and it will not know what data point to reconstruct. So you want there to be no gaps so that any point you sample in that z-space will map to a valid data point x according to your distribution of training points x. OK. That's the intuition I have.

OK, so in the last five or 10 minutes, I want to talk about what kinds of representations are learned by a VAE and other generative models. And so we're going to go back to this example here, and we're going to ask, are the latent variables z that the VAE learns actually related to what I gave as an intuition? I said I could write a Python program that takes a normal distribution, maps it to grass color; a Bernoulli distribution, maps it to the turns of the river. Well, is that what actually happens when I train a model without hand-defining it?

OK, so I will take my data, I'll learn my encoder and my decoder. My decoder can generate samples, and this is a good generative model of my data. But now let's look at the encoder. Is the encoder separately useful? Is a VAE a good representation learner?

OK, so we'll visualize it like this. We're going to take our latent space and we're going to move in axis line directions in the latent space and see what images those map to. OK, so if I move along one axis of latent space, it ends up causing this kind of smooth variation in the output space where the curvature of the river changes. If I move along another dimension of latent space, it ends up changing the color of the grass. Oh, sorry. OK.

So, indeed, my learned latent distribution ends up having different dimensions in latent space controlling different types of visual variation on this data. And those types of visual variation actually do relate to the underlying parameters that define causally how the data was generated. Like, there is a causal process, a Python program I wrote that takes these attributes and turns them into rivers and grass and so forth.

And one of those attributes is the grass color is a normal distribution. So there's one dimension of the latent space that generated the data that should control grass color, and the VAE actually recovers that. So it identifies that true factor of-- that true causal factor in the data.

OK, but this is not always going to be the case. The other factor of variation that I'm showing you here, which is the curvature of the river, actually does not correspond to any of the underlying variables in my Python code. Because the Python code took a vector of Bernoulli coin flips to determine the shape of the river, and there was no single variable, no Gaussian variable that defined the overall shape of the river, it was just a sequence of Bernoulli steps.

But somehow, the VAE learned this abstracted, intuitive physics of the world. It didn't actually learn the physics of the world that I wrote in Python to generate this data, it came up with this abstracted abstraction, which is that there should be a factor of variation in this data that controls the curvature of the river that's a good explanation.

So a VAE, like other representation learners, can learn different independent factors of variation that affect the primary independent-- it can learn different latent variables, different dimensions of latent space that control different independent factors of variation-- we call that disentanglement, that if I vary the river or curvature dimension, it does not affect the color, and if I vary the color, it does not affect the river curvature dimension.

This is a disentangled representation where there's independent-- independence between the effects of those two factors, but those two factors don't necessarily map on to actually the causal mechanisms underlying how this world was defined. They might be more abstracted from it.

OK. So VAEs are not the only models that have this property of learning organized latent spaces that are good maps or good representations of the world. In fact, this is an example of a GAN, and it does-- has just the same types of properties.

So we covered GANs very briefly last lecture, and it's not-- GANs didn't have any encoder that maps from data to latents. So VAEs do have an encoder, they can go from data to the representation of the data in this disentangled space. Again, doesn't have that, but nonetheless, the latent variables, if you sample random latent variables or sample along a grid of latent variables, they do show that they actually model structure in a similar way to the VAE.

And here's just an example from a model called BigGAN where there's one direction in latent space that controls the orientation of the bird and one direction that controls the background color.

And you'll see this over and over again. In all the types of generative models that we have talked about, there are some kind of latent variables usually, like in diffusion models, maybe the noise. Or there are other intermediate variables like the activations of the network as it processes-- as a decoder processes the data. The activations on some layer, I can look at directions in that activation space.

And these tend to learn structured representations of the data. That is the mechanism by which the model can fit to the world effectively. It has to learn some kind of factors that can parsimoniously explain the data. If it only has so much capacity, so many dimensions, this is a kind of somehow pops out that this is a good representation of the world that allows you to generate photos that maximize likelihood.

OK. So generative modeling forces you to explain the world. Explaining the world is done well by actually capturing independent components and things that we have physical names for, like background color, but sometimes things are more abstracted, like low-frequency curvature of the river.

Yeah, this one maybe just a kind of fun visualization to end on. This is, again, that same map. And here, I'm coloring the input points according to their coordinates within this space. And here's what happens as I train it. So what you can see is that this thing gets-- the points in this space get remapped to the sphere, but not like in a disorganized way. It's smooth, it's a smooth map.

And that's kind of why-- it's meant to give an intuition for how these dimensions of the latent space might map to smooth properties of the input space, because on this simple example, all it's really doing is squishing the thing down.

But there's also an important point, which is that it didn't quite fill this gap here. And if it really was going to squish it into a sphere, a Gaussian, then it should fill that. And you'll see this when you're training generative models, that in the latent space, things will be smooth and nice and there will be a seam because it's like you're mapping this complicated topology onto a sphere, and you're going to have cuts and seams, and you see it right here.

So if you're navigating through VAE latent space or diffusion model activation space on some layer of the network, you'll come across these weird jumps where it's smooth moves, nice variation, and then suddenly it switches to a different category, and that's what's going on here. OK.

Also, these tendrils. Like, you can't-- this is all smooth transformations. How do you cut? You can't really cut, so you get this tendril of probability.

OK, so these are the models that we've seen in the last two lectures. We covered a lot. So we talked about a bunch. I'd say I think the most popular right now is diffusion models. The most complete, like full picture, that encompasses everything is VAEs. The most cute and fun is GANs. The ones that are popular in language modeling is autoregressive.

And they all have different trade-offs. Some of them have latent variables that are explicit, some don't. Some give you a density or energy function, some don't. And some give you a way of sampling efficiently and some don't. So that's for your own reference to review. OK, thank you.