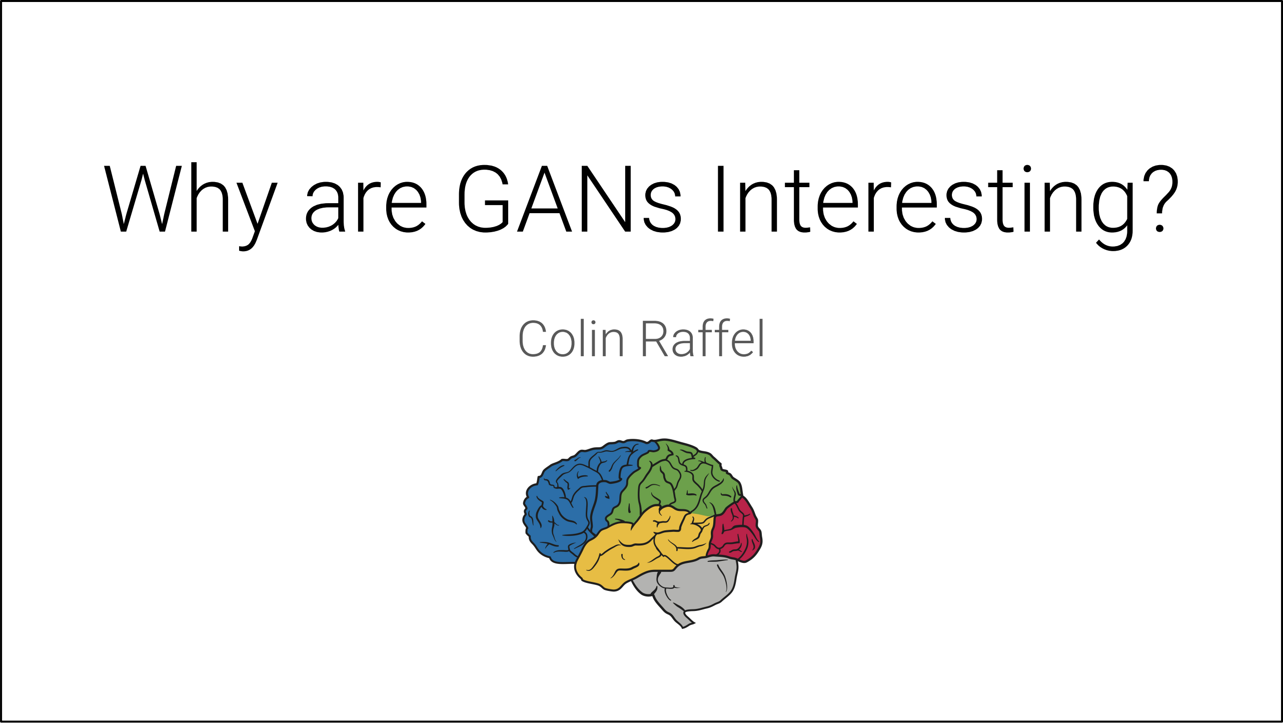
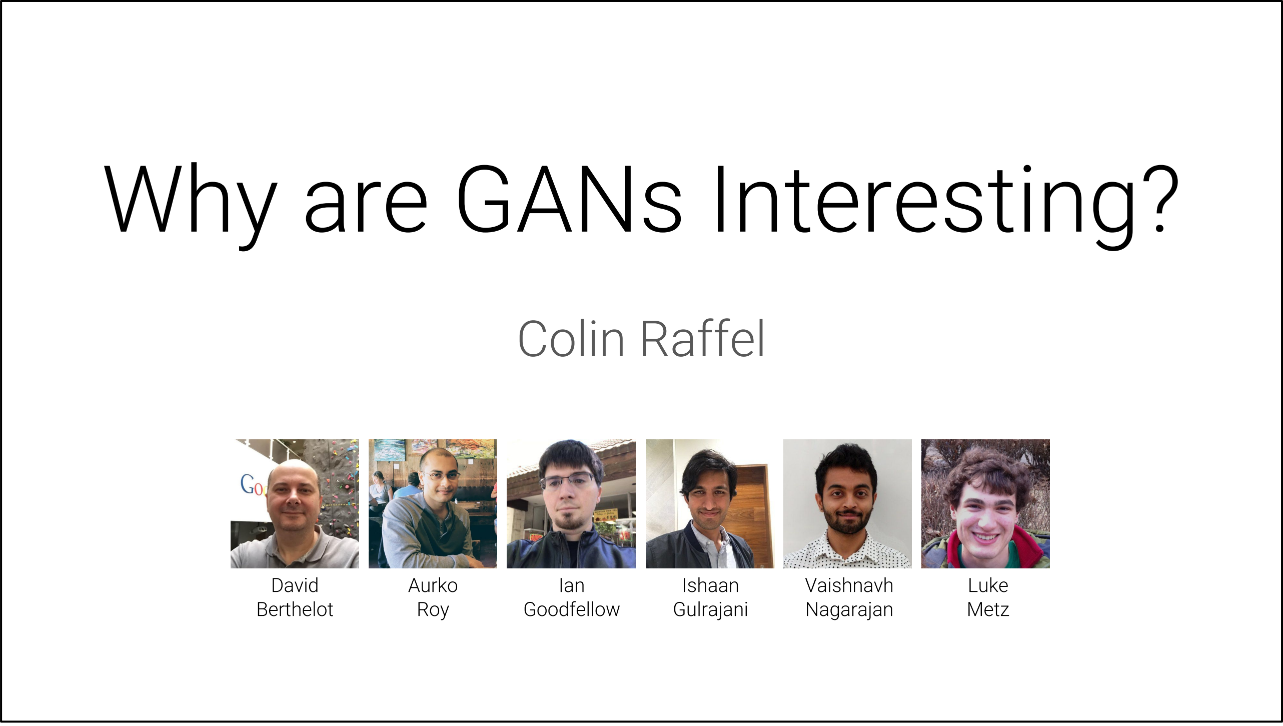


Why are GANs Interesting?

Colin Raffel

You are viewing the PDF version of these slides; any animations will appear as static images.



Why are GANs Interesting?

Colin Raffel

David Berthelot

Aurko Roy

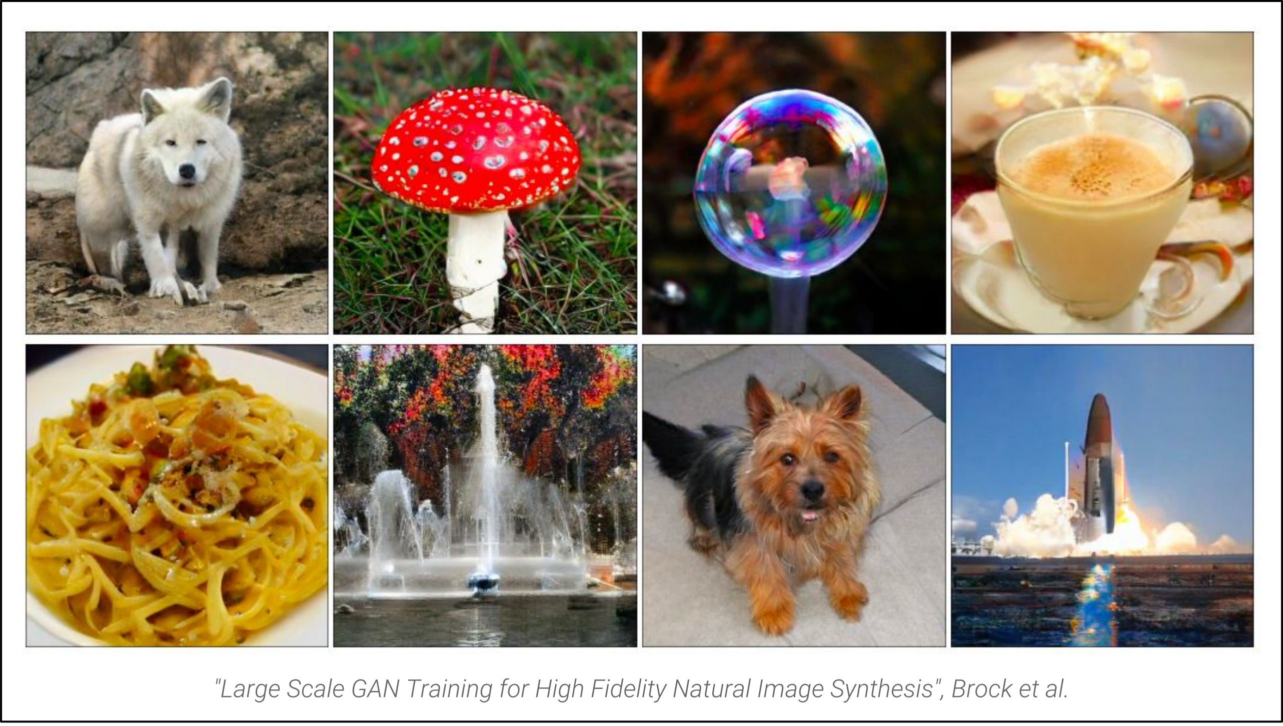
Ian Goodfellow

Ishaan Gulrajani

Vaishnavh Nagarajan

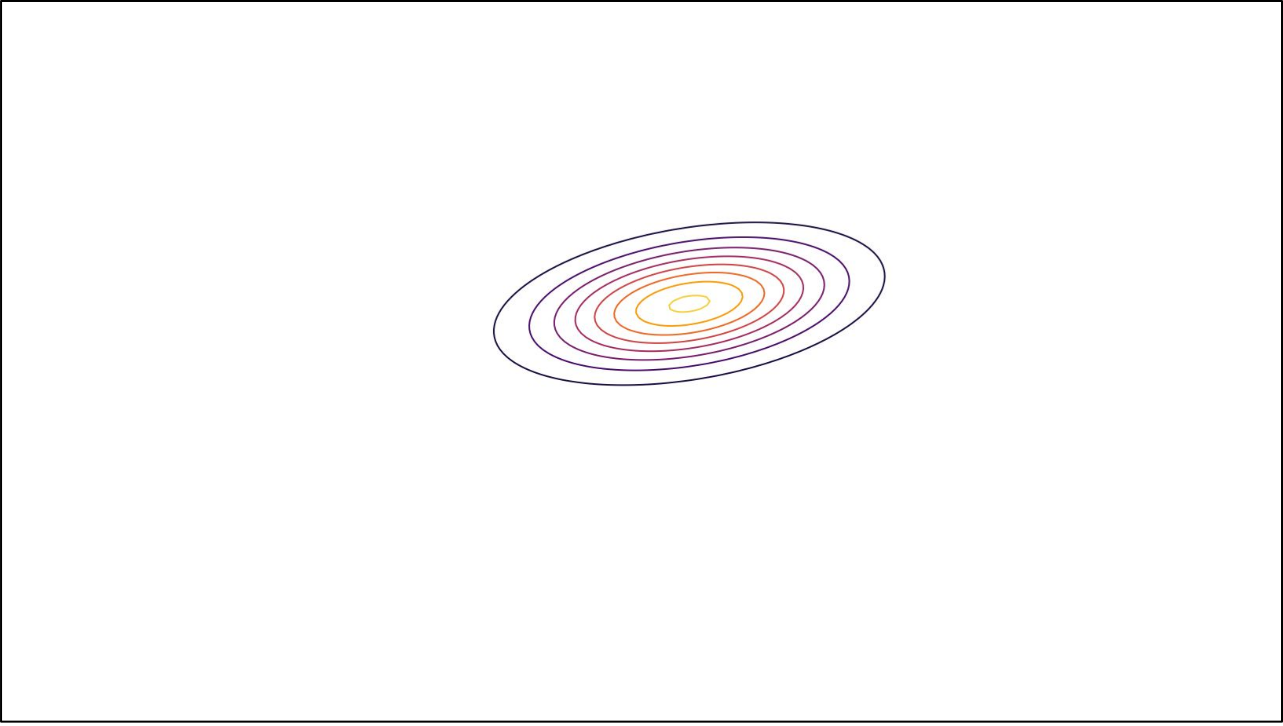
Luke Metz

If you are confused about something, it's probably because I haven't explained it well and other people are probably confused too, so please feel free to stop me to ask questions. If something I'm describing seems like a bad idea or is not well-motivated, please let me know and I'll try to clarify.

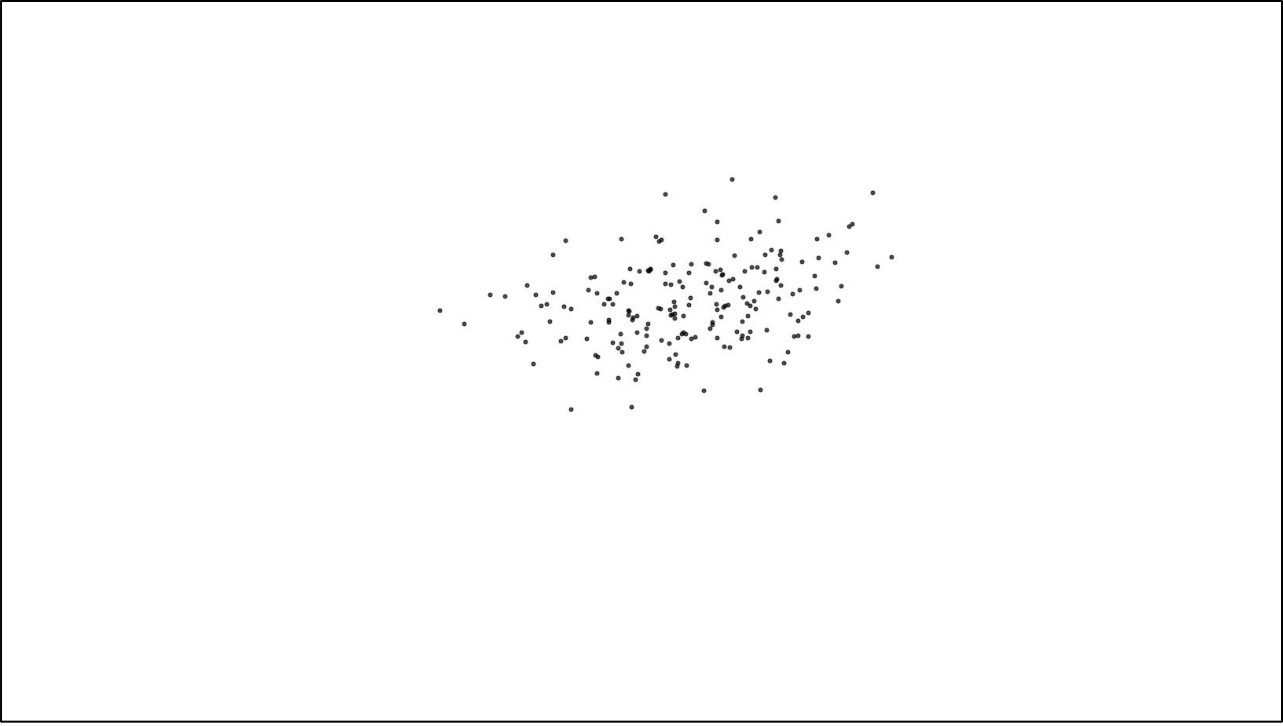


"Large Scale GAN Training for High Fidelity Natural Image Synthesis", Brock et al.

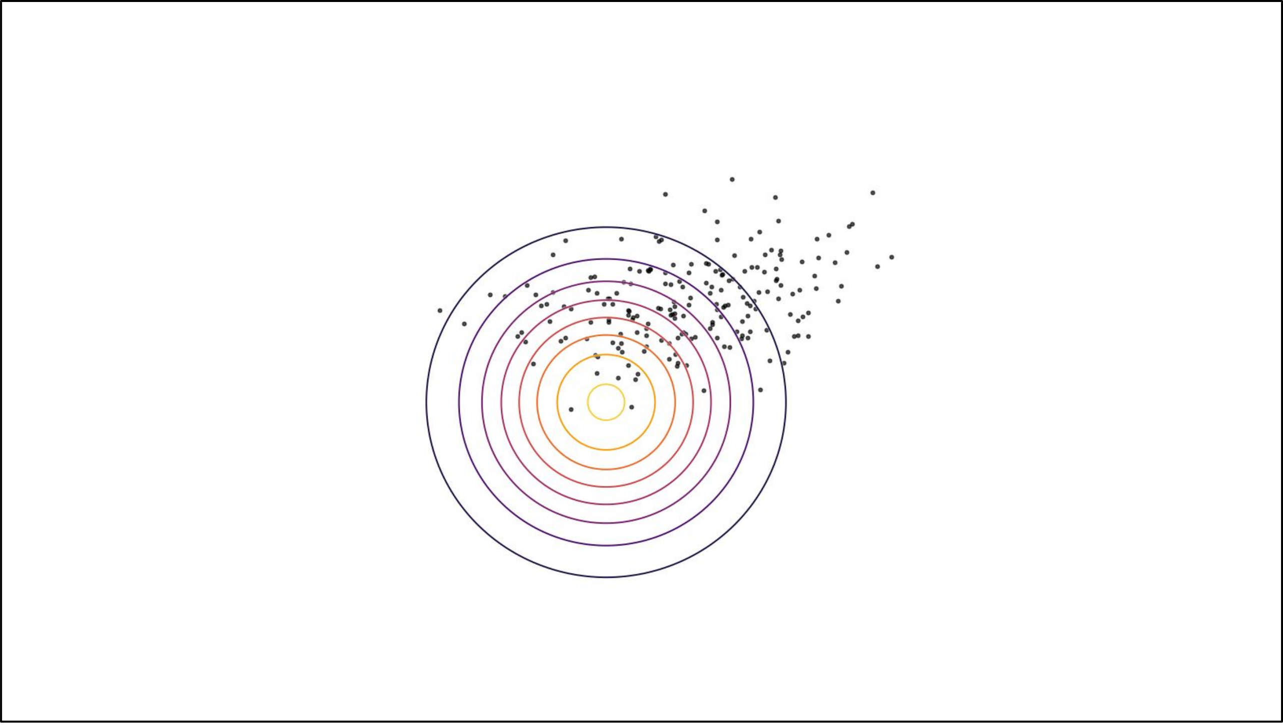
This is what most people think of when they think of GANs. They're great at generative modeling of images - we can train them on large image datasets and they produce new made-up images which look quite realistic. This is really cool and impressive, but I think there are lots of other reasons"GANs are interesting, which is what I'll focus on today.



Let's take a step back and talk about generative modeling. We assume there's some underlying probability distribution that we want to model. For the moment let's consider modeling this 2D Gaussian distribution. This is a toy example; in practice we want to model extremely complex distributions in high dimensions, such as the distribution of natural images.



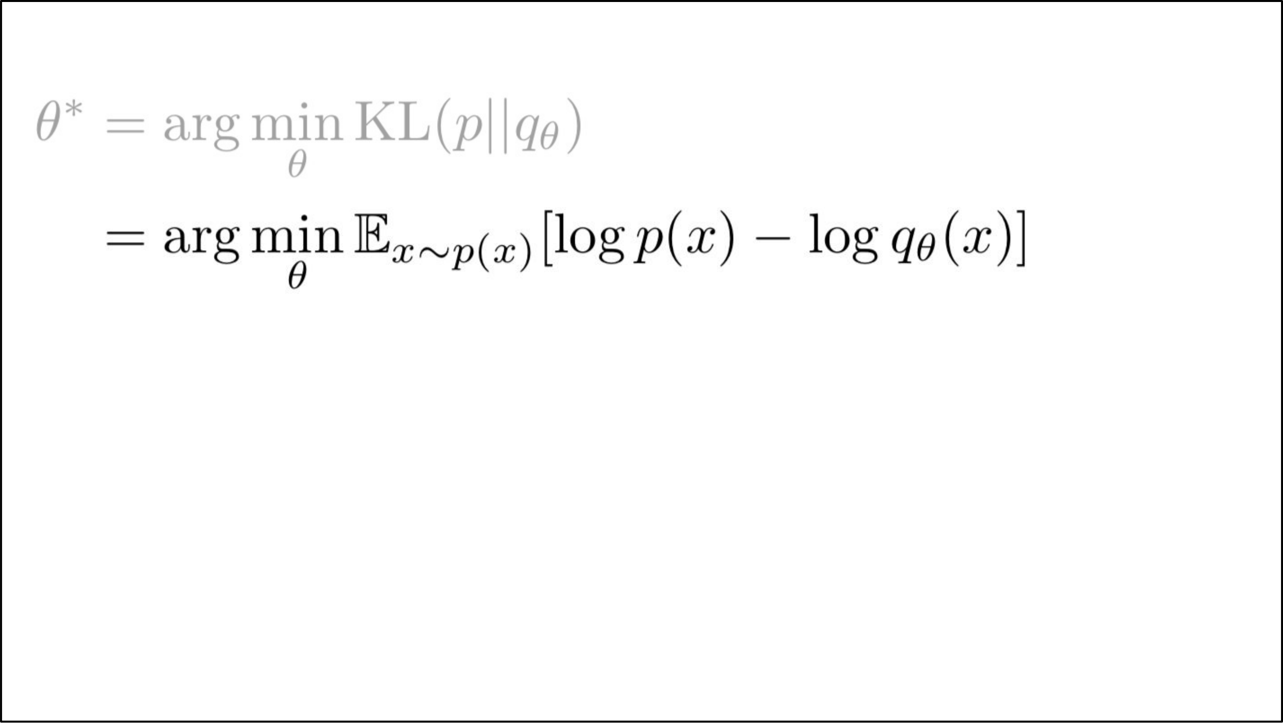
Of course, we don't have access to the true distribution. Instead, we have access to some samples from it. Shown here are some samples from our true Gaussian distribution. We want to be able to recover the true distribution using these samples.



Let's try to recover the true distribution using a 2D gaussian. To do so, we'll need to estimate the parameters (mean and covariance) of this Gaussian.

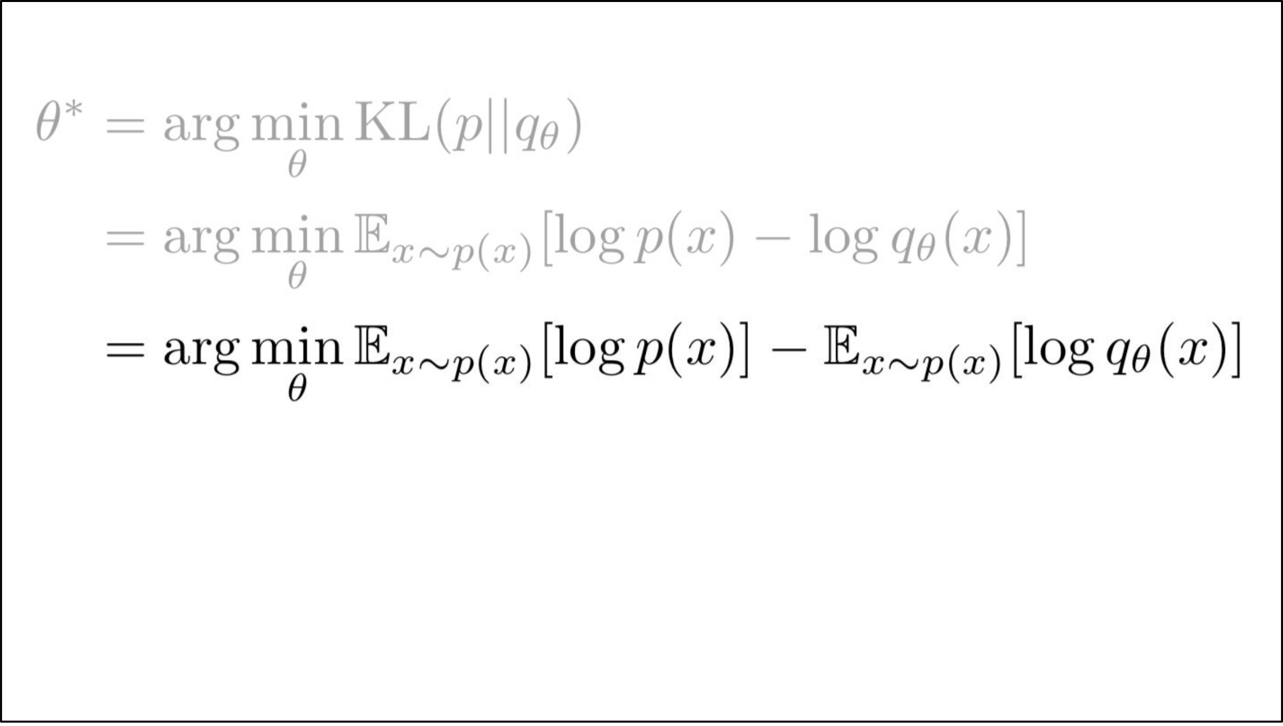
$$\theta^* = \arg\min_{\theta} \mathrm{KL}(p||q_\theta)$$

To optimize these parameters, let's define a loss function. Why don't we try minimizing the KL divergence between the true distribution p and the estimated distribution q_theta. Our chosen parameters will be those which minimize the KL divergence between p and q_theta.



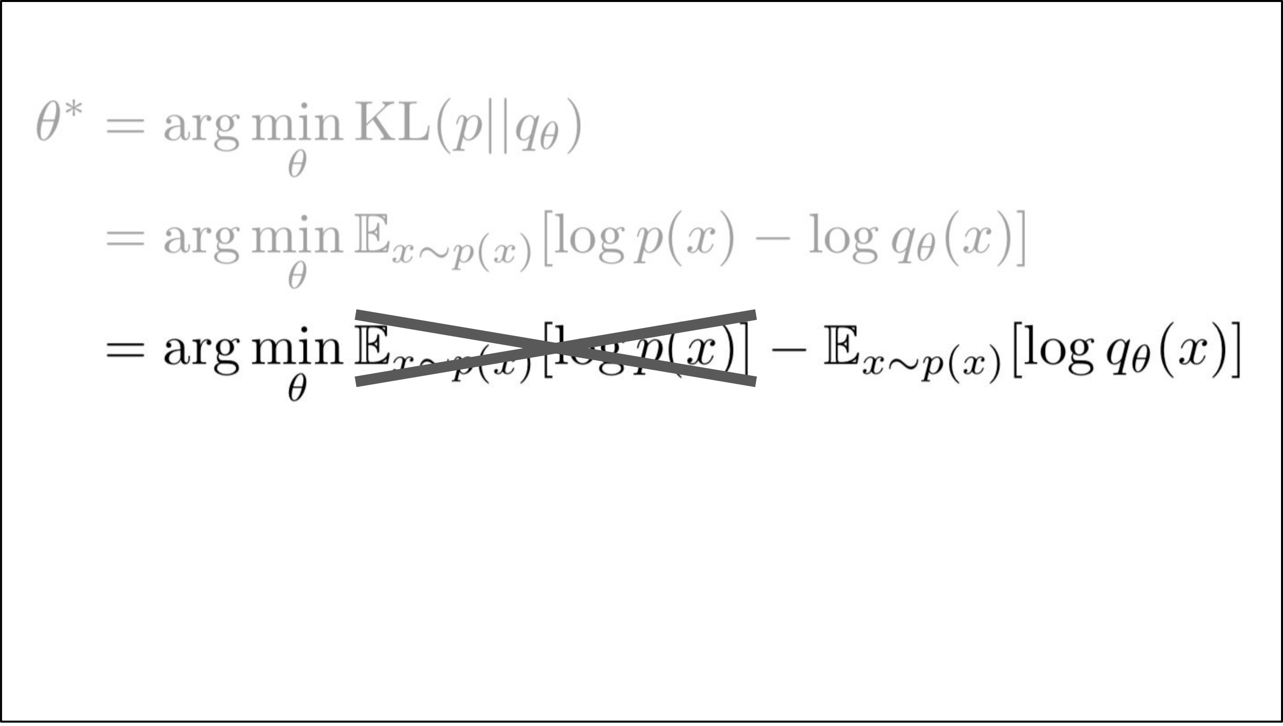
$$
\begin{aligned}
\theta^* &= \arg\min_\theta \mathrm{KL}(p||q_\theta) \\
&= \arg\min_\theta \mathbb{E}_{x\sim p(x)}[\log p(x) - \log q_\theta(x)]
\end{aligned}
$$

First, let's expand out the definition of the KL divergence.



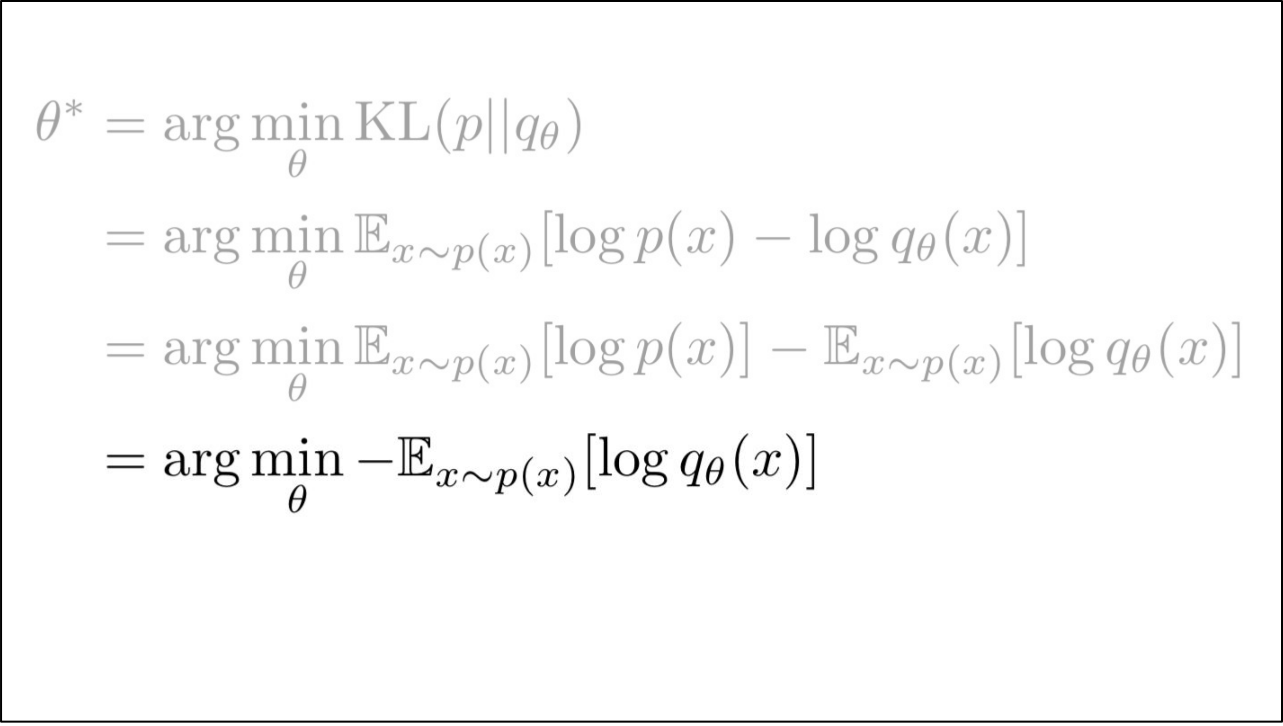
$$
\begin{aligned}
\theta^* &= \arg\min_\theta \mathrm{KL}(p||q_\theta) \\
&= \arg\min_\theta \mathbb{E}_{x\sim p(x)}[\log p(x) - \log q_\theta(x)] \\
&= \arg\min_\theta \mathbb{E}_{x\sim p(x)}[\log p(x)] - \mathbb{E}_{x\sim p(x)}[\log q_\theta(x)]
\end{aligned}
$$

Now, let's separate the two terms into two separate expectations.



$$\begin{aligned}\theta^* &= \arg\min_\theta \mathrm{KL}(p||q_\theta) \\ &= \arg\min_\theta \mathbb{E}_{x\sim p(x)}[\log p(x) - \log q_\theta(x)] \\ &= \arg\min_\theta \xcancel{\mathbb{E}_{x\sim p(x)}[\log p(x)]} - \mathbb{E}_{x\sim p(x)}[\log q_\theta(x)]\end{aligned}$$

The first expectation is just the entropy of the true distribution. Changing theta doesn't change this term, so we can ignore it.

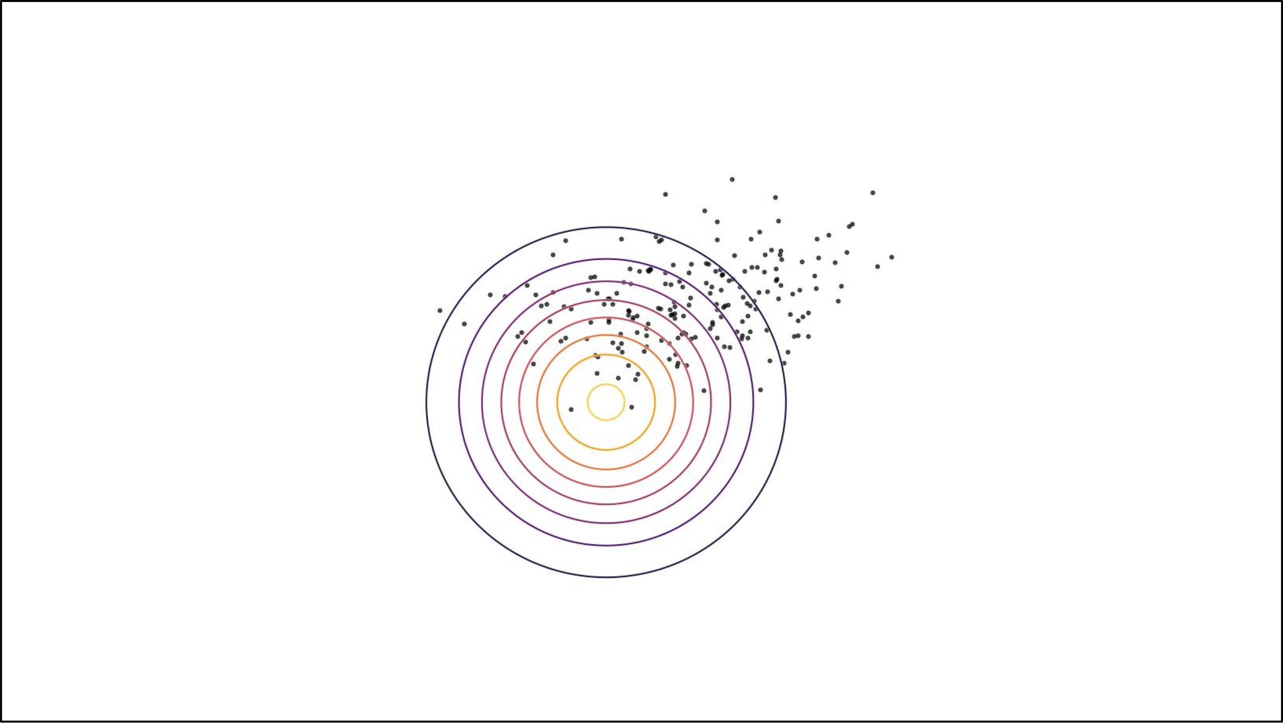


$$\begin{aligned}\theta^* &= \arg\min_\theta \mathrm{KL}(p||q_\theta)\\ &= \arg\min_\theta \mathbb{E}_{x\sim p(x)}[\log p(x) - \log q_\theta(x)]\\ &= \arg\min_\theta \mathbb{E}_{x\sim p(x)}[\log p(x)] - \mathbb{E}_{x\sim p(x)}[\log q_\theta(x)]\\ &= \arg\min_\theta -\mathbb{E}_{x\sim p(x)}[\log q_\theta(x)]\end{aligned}$$

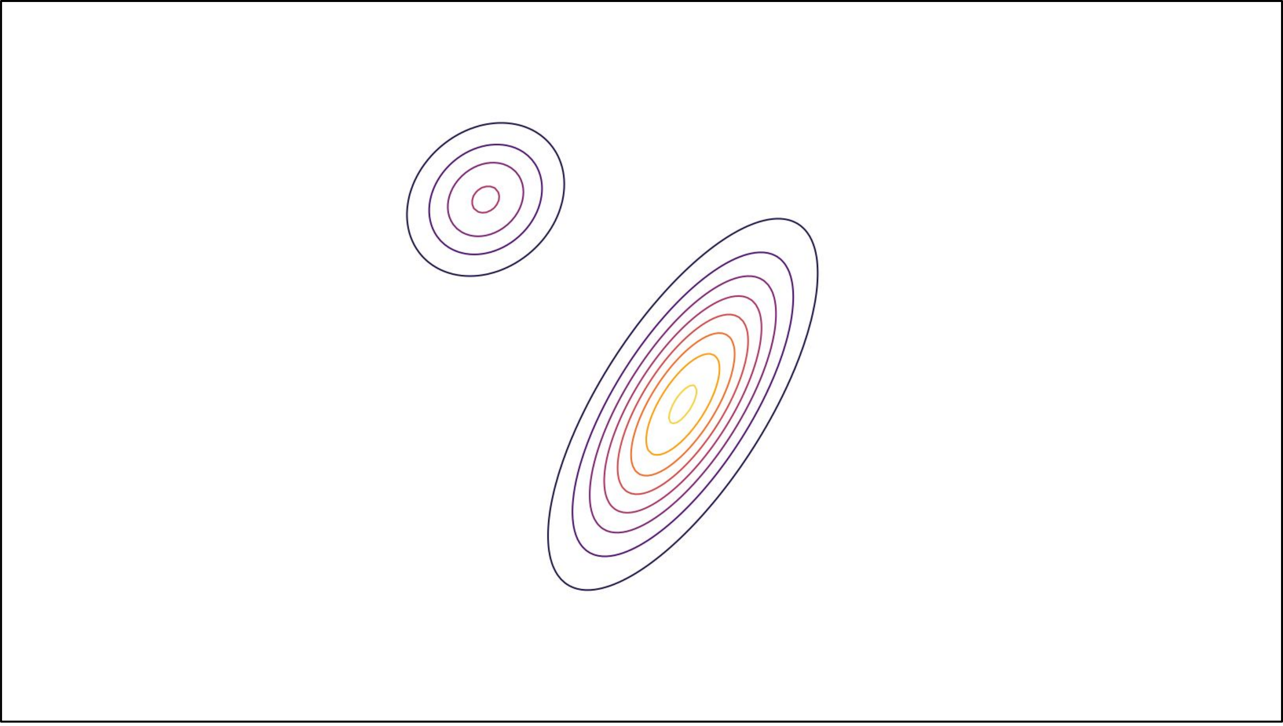
So now we're left with this term - we're minimizing the negative log probability of samples from p under the distribution q_theta.

$$
\begin{aligned}
\theta^* &= \arg\min_\theta \mathrm{KL}(p||q_\theta) \\
&= \arg\min_\theta \mathbb{E}_{x\sim p(x)}[\log p(x) - \log q_\theta(x)] \\
&= \arg\min_\theta \mathbb{E}_{x\sim p(x)}[\log p(x)] - \mathbb{E}_{x\sim p(x)}[\log q_\theta(x)] \\
&= \arg\min_\theta -\mathbb{E}_{x\sim p(x)}[\log q_\theta(x)] \\
&= \arg\max_\theta \mathbb{E}_{x\sim p(x)}[\log q_\theta(x)]
\end{aligned}
$$

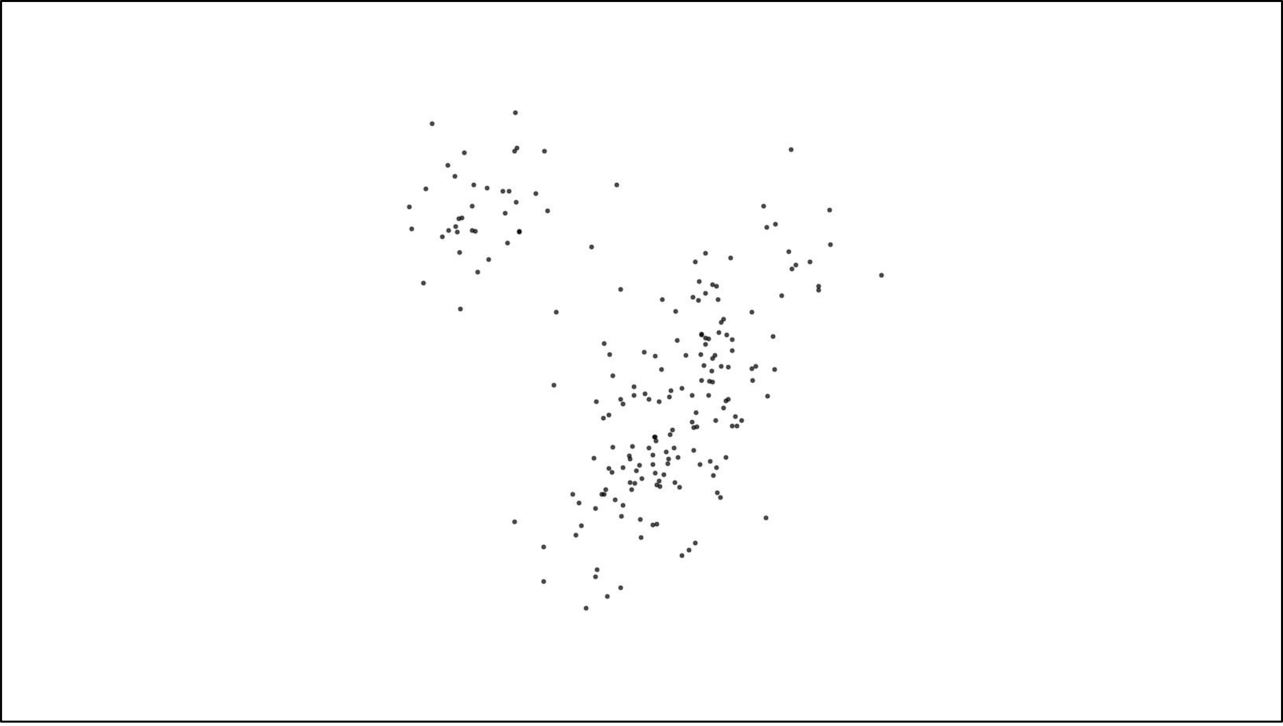
Now, if we remove the negative and switch to a maximization instead of a minimization, we get something which might look familiar - the equation for maximum-likelihood estimation. We're trying to maximize the likelihood of samples from p under our model q_theta. So, all of this was a long-winded way of showing how minimizing the KL divergence is equivalent to maximum likelihood estimation. This may be review for you, but I wanted to make sure we were all on the same page.



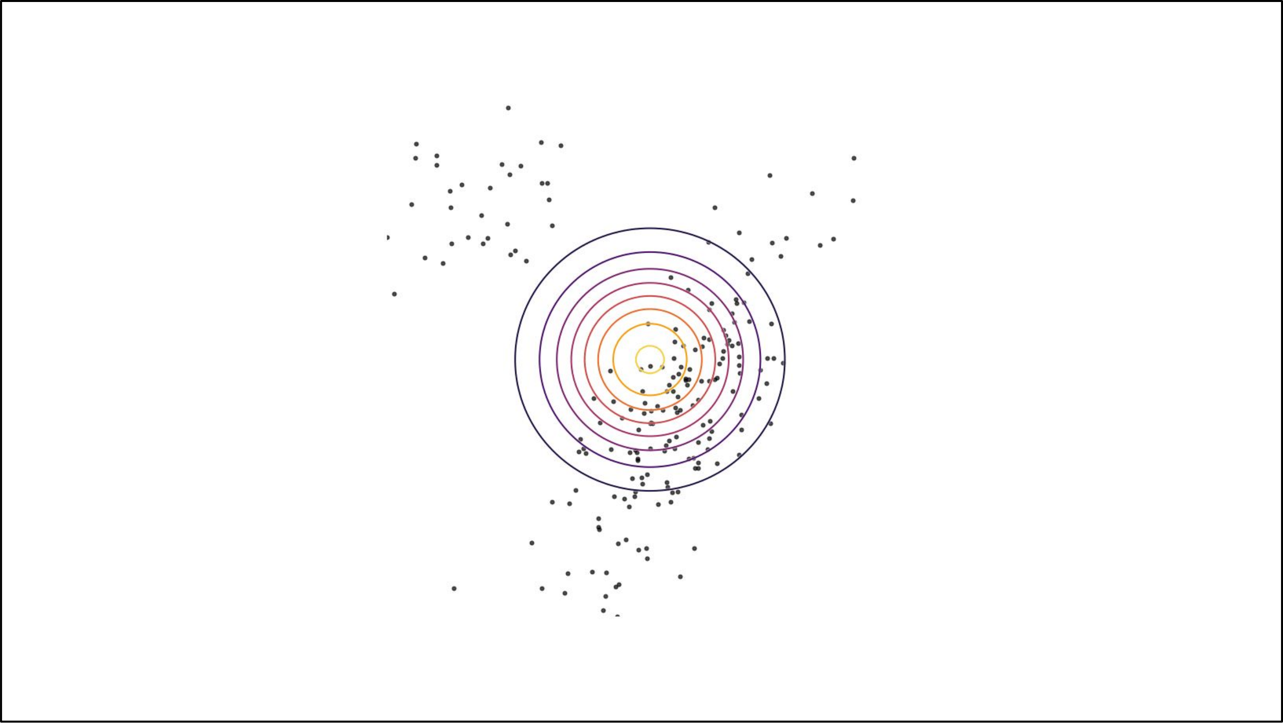
Ok, let's do it - let's minimize the KL divergence between p and q_theta, or equivalently maximize the likelihood of samples from p under q_theta, using gradient descent. This is what happens. it totally works! Cool. Maximum likelihood is a super common, and pretty old, way of fitting generative models, and in some cases it does what we want.



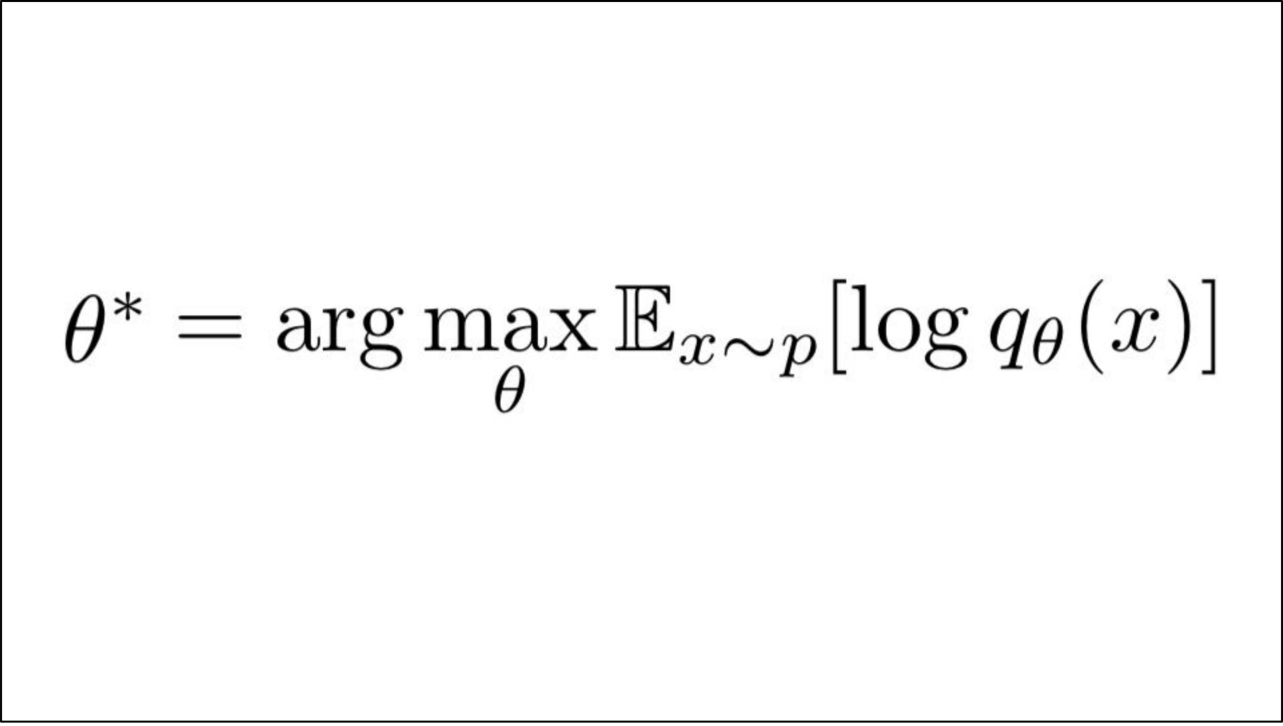
Ok, let's make things a little harder. Let's try to fit the distribution shown here - this is a mixture of two Gaussians.



Again, we're going to try to do so using only samples from the true distribution.

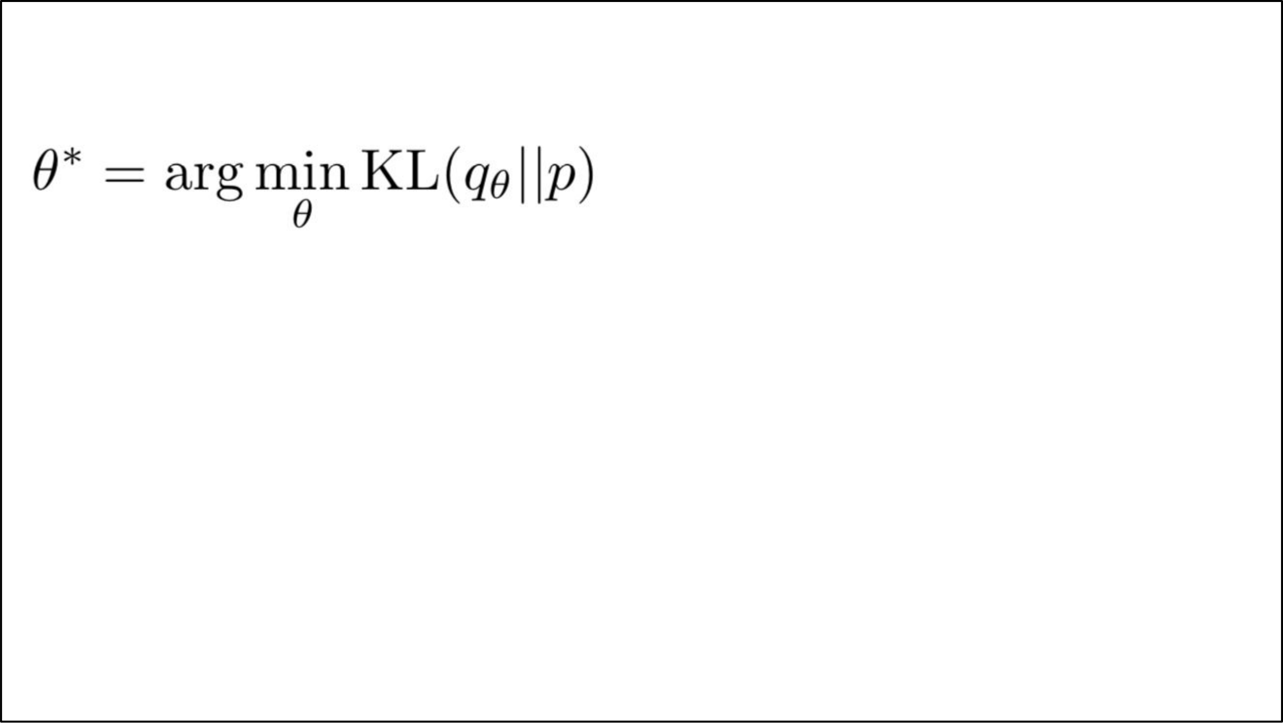


So, this is what happens when we try to fit a 2D Gaussian to our mixture distribution using maximum likelihood. In this case, our estimated is "misspecified". The distribution just kind of spreads out to try to cover both modes. The result is that a lot of the probability mass of our estimated distribution actually covers regions of space which have extremely low probability under the true distribution. In other words, the samples from this distribution may not be very realistic.



$$\theta^* = \arg\max_{\theta} \mathbb{E}_{x \sim p}[\log q_\theta(x)]$$

So why does this happen? Let's look at the maximum likelihood equation again. What happens if we draw a sample from p and it has low probability under q? As q theta x approaches zero, log q theta x goes to negative infinity. So in other words, it's really really bad if we draw a sample from p and q_theta assigns a low probability to it. The result is that the estimated model tries to cover the entire domain of the true distribution, even if it means assigning probability mass to "unrealistic" regions of space.



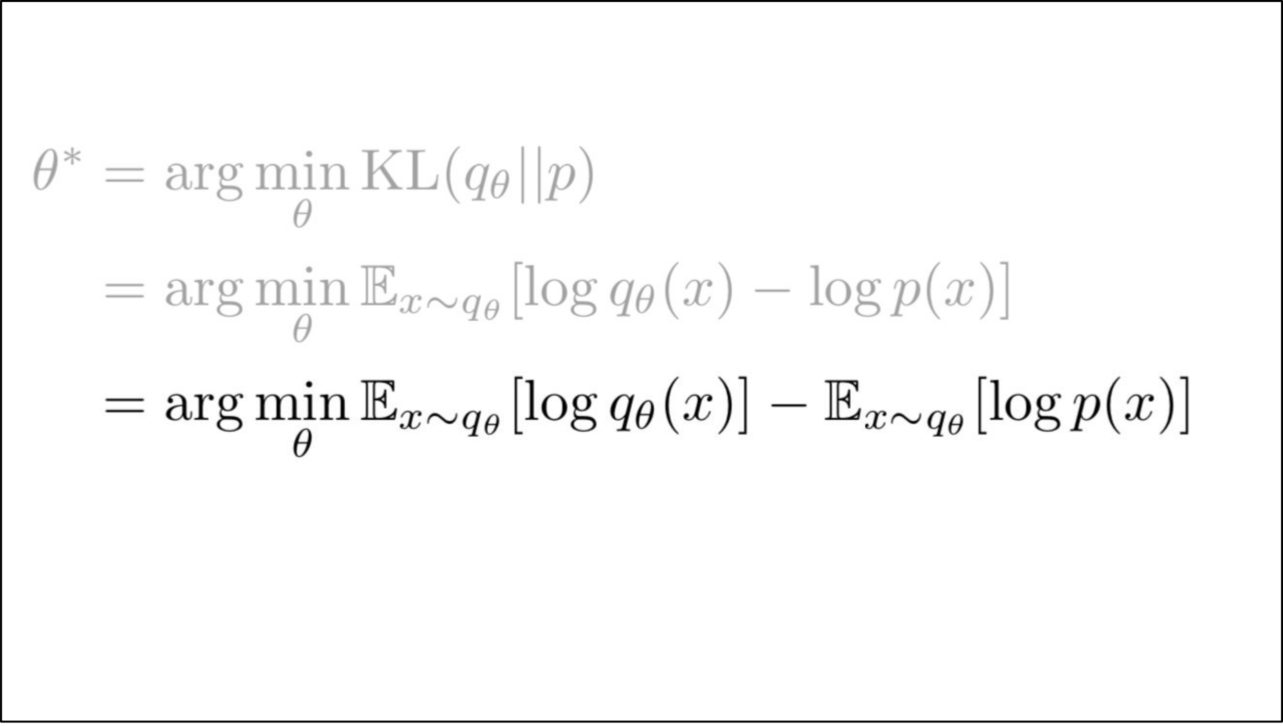
$$\theta^* = \arg\min_\theta \mathrm{KL}(q_\theta || p)$$

Let's try something different. Instead of minimizing the KL divergence between p and q_theta, let's try minimizing the KL divergence between q_theta and p - the "reverse KL".

$$\theta^* = \arg\min_\theta \mathrm{KL}(q_\theta || p)$$

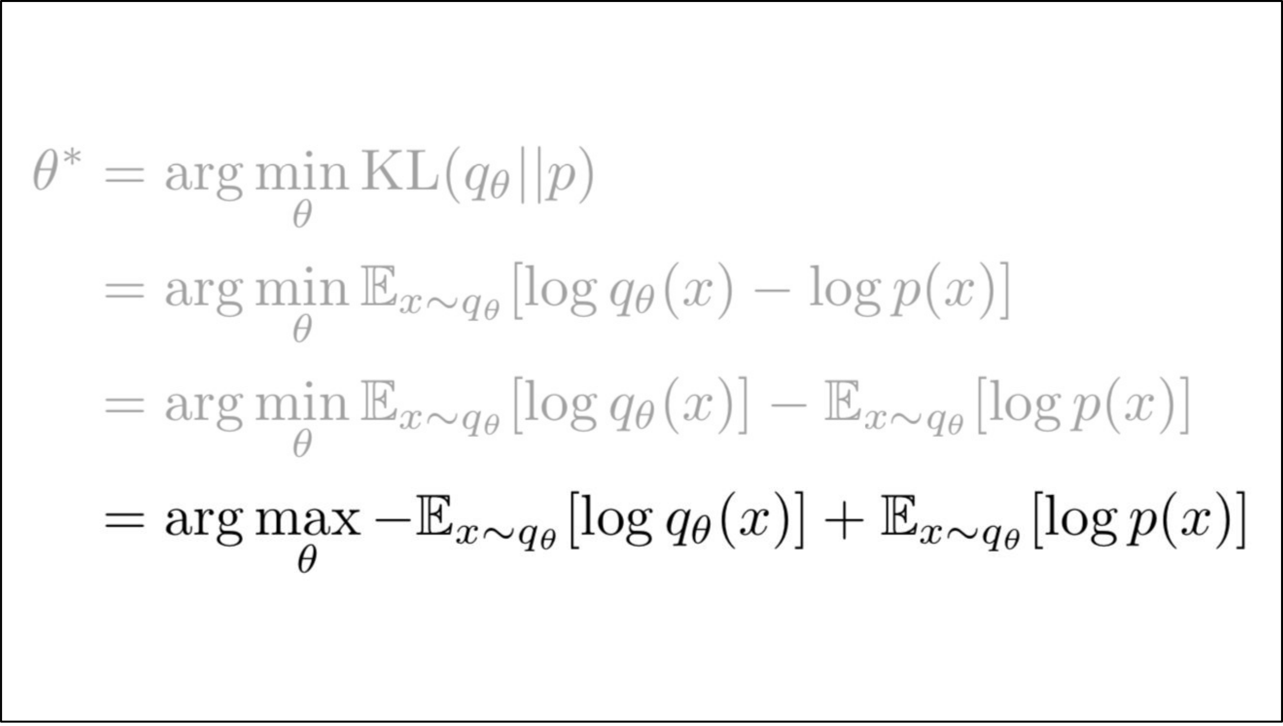
$$= \arg\min_\theta \mathbb{E}_{x\sim q_\theta}[\log q_\theta(x) - \log p(x)]$$

First, expanding out the definition of the KL divergence again….



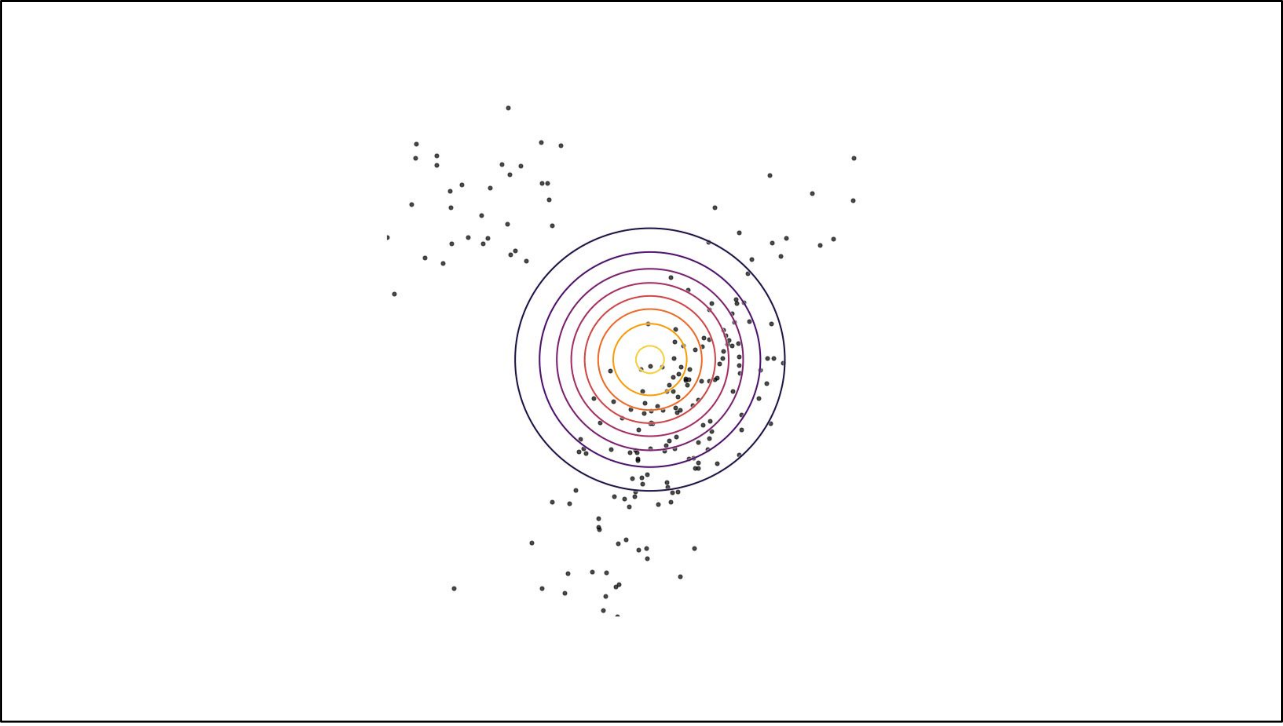
$$\begin{aligned}\theta^* &= \arg\min_\theta \mathrm{KL}(q_\theta || p) \\ &= \arg\min_\theta \mathbb{E}_{x\sim q_\theta}[\log q_\theta(x) - \log p(x)] \\ &= \arg\min_\theta \mathbb{E}_{x\sim q_\theta}[\log q_\theta(x)] - \mathbb{E}_{x\sim q_\theta}[\log p(x)]\end{aligned}$$

And, separating out the two expectations...

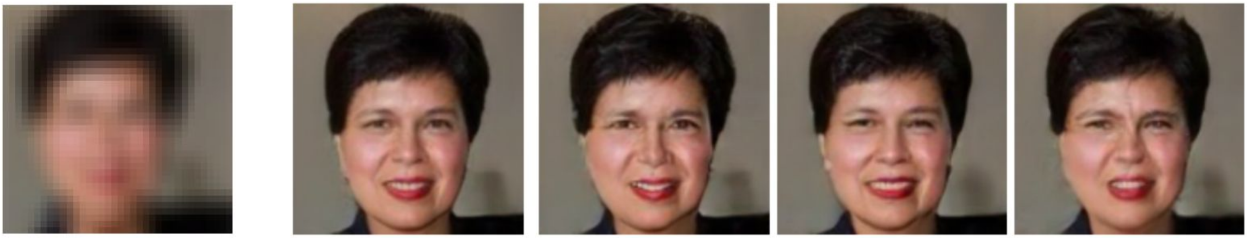


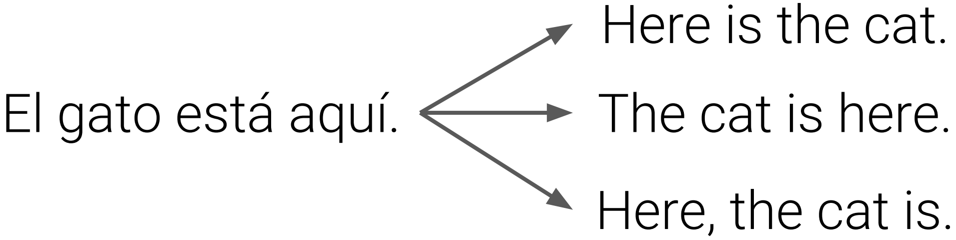
$$\begin{aligned}
\theta^* &= \arg\min_{\theta} \mathrm{KL}(q_\theta || p) \\
&= \arg\min_{\theta} \mathbb{E}_{x \sim q_\theta}[\log q_\theta(x) - \log p(x)] \\
&= \arg\min_{\theta} \mathbb{E}_{x \sim q_\theta}[\log q_\theta(x)] - \mathbb{E}_{x \sim q_\theta}[\log p(x)] \\
&= \arg\max_{\theta} -\mathbb{E}_{x \sim q_\theta}[\log q_\theta(x)] + \mathbb{E}_{x \sim q_\theta}[\log p(x)]
\end{aligned}$$

and finally, swapping the min for a max and negating the two terms, here's what we get. These two terms also have an intuitive explanation. The first term is simply the entropy of q theta. So, we want to find a distribution q theta which has high entropy, so its probability mass is spread out. The second term is the log probability of samples from q_theta under the true distribution p. So in other words, any sample from q_theta has to be realistic according to our true distribution.



So, what happens when we optimize the reverse KL instead of the KL? Our model basically picks a single mode, and models it well! Why? Well, the resulting distribution is reasonably high-entropy. And, any sample from the estimated distribution has a reasonably high probability under p, because the support of q_theta is basically a subset of the support of p. The drawback here is that we are basically missing an entire mode of the true distribution.





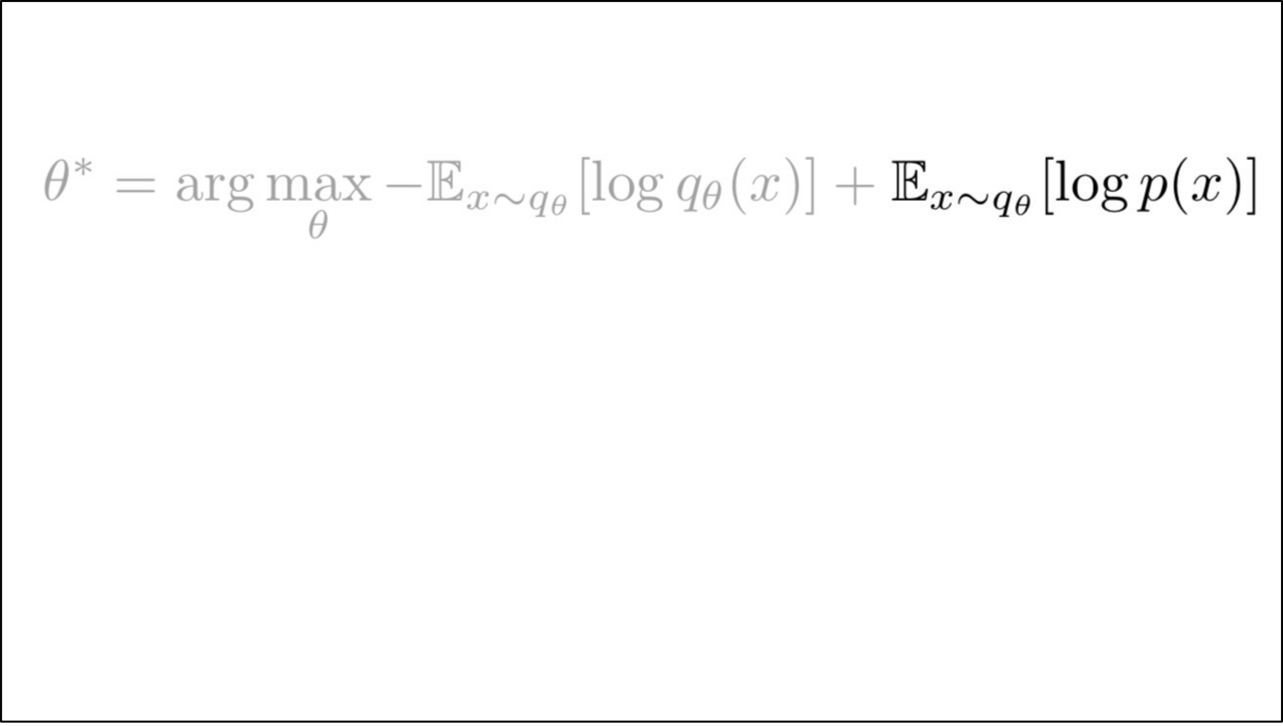
El gato está aquí.

Here is the cat.

The cat is here.

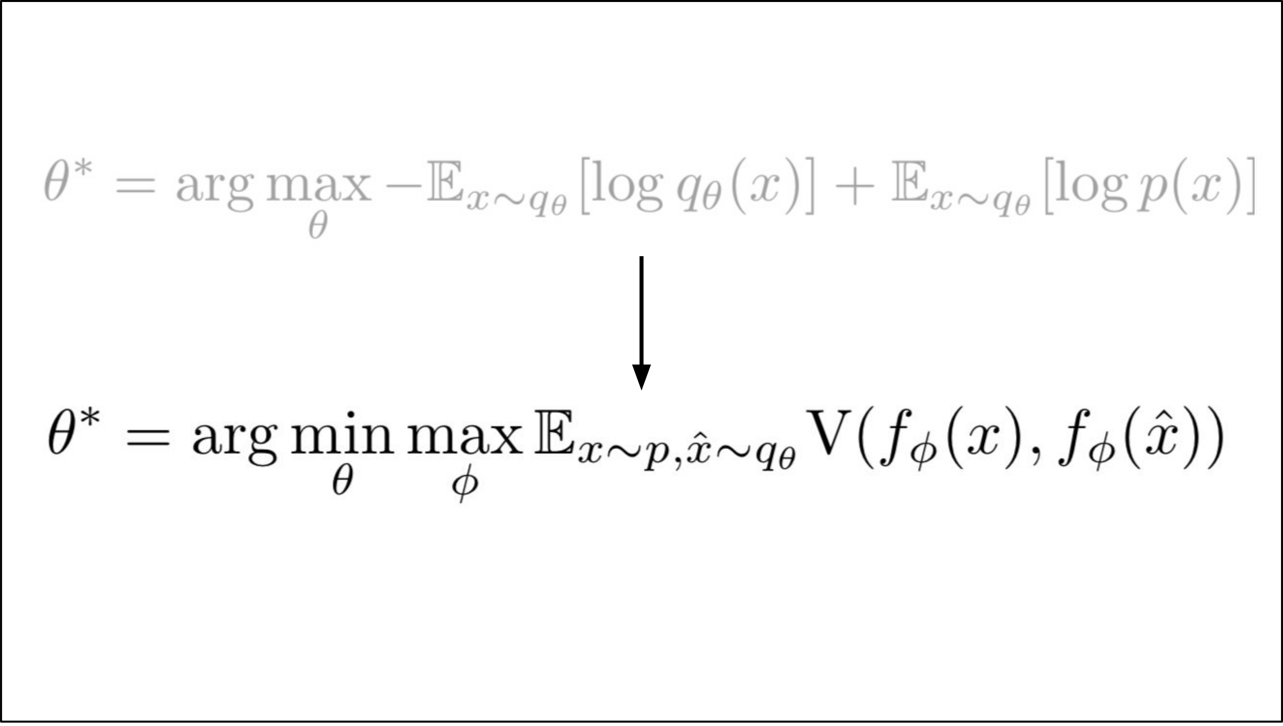
Here, the cat is.

When might this be a desirable solution? Consider conditional generative modeling tasks, like the two shown here - on the top, we have a low resolution image and we are modeling the distribution over the super-resolved version. This figure was made by my colleague David Berthelot. On the bottom, we are modeling the distribution of possible translations of an input sentence. In both cases there are multiple possible "good" solutions. In practice, it may be much more important that our model produces a single high-quality output than that it correctly models the distribution over all possible outputs. In this case, maximum likelihood is arguably worse in practice because it might produce low-quality or incorrect outputs.



$$\theta^* = \arg\max_\theta -\mathbb{E}_{x\sim q_\theta}[\log q_\theta(x)] + \mathbb{E}_{x\sim q_\theta}[\log p(x)]$$

So, I've kind of cheated here. We can't actually compute the term I've highlighted here in practice because it requires evaluating the true probability of a sample, and we don't have access to the true distribution, we only have access to samples from it.. So, we can't actually use reverse KL to optimize the parameters of our estimated distribution.

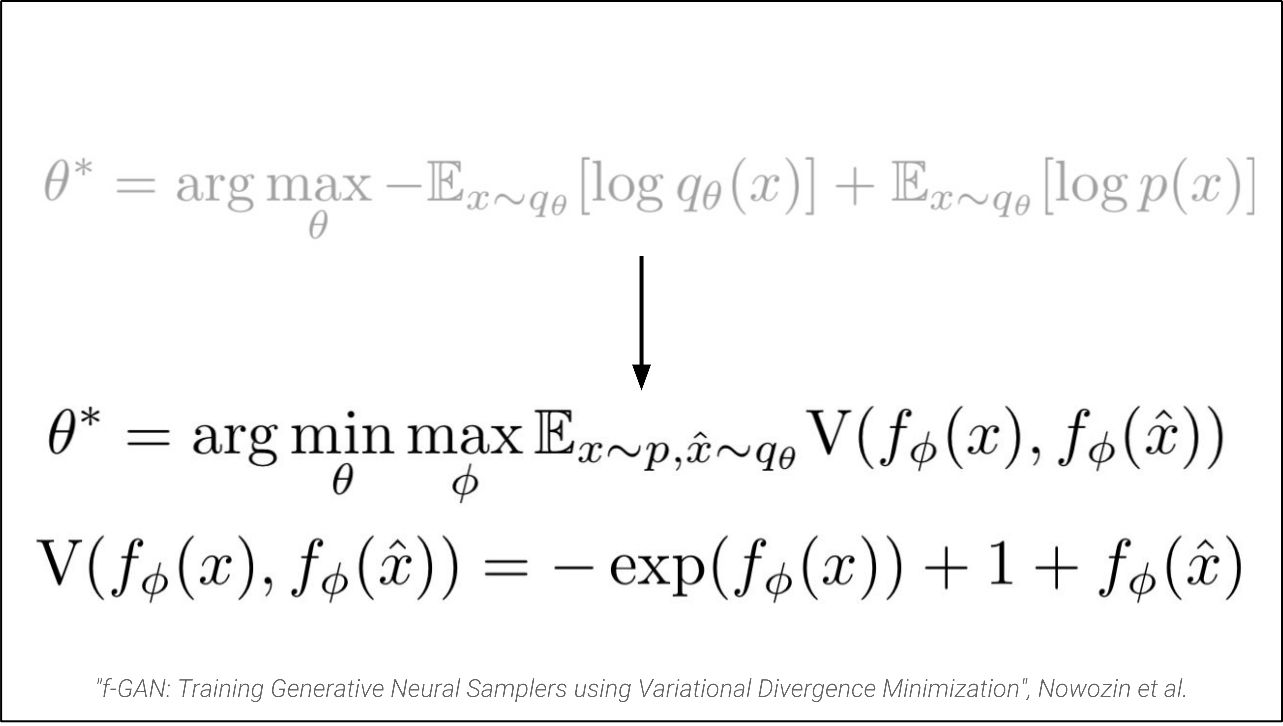


$$\theta^* = \arg\max_{\theta} -\mathbb{E}_{x\sim q_\theta}[\log q_\theta(x)] + \mathbb{E}_{x\sim q_\theta}[\log p(x)]$$

$$\Big\downarrow$$

$$\theta^* = \arg\min_{\theta}\max_{\phi}\mathbb{E}_{x\sim p,\hat{x}\sim q_\theta}\mathrm{V}(f_\phi(x), f_\phi(\hat{x}))$$

So, here's where generative adversarial networks come in. GANs replace the divergence between two distributions with what we can think of as a learned divergence. This amounts to learning a discriminator or critic network which is written here as f phi. f_phi takes in samples from the true distribution and from the learned distribution and outputs a scalar value. Then, we define some loss function V which takes in outputs of the discriminator for real and generated data and outputs a value to optimize. We optimize the parameters of the generator and the discriminator together in a two-player game, where the discriminator's parameters phi are adjusted to maximize this loss function and the parameters of the estimated distribution are optimized to minimize it.



$$\theta^* = \arg\max_{\theta} -\mathbb{E}_{x\sim q_\theta}[\log q_\theta(x)] + \mathbb{E}_{x\sim q_\theta}[\log p(x)]$$

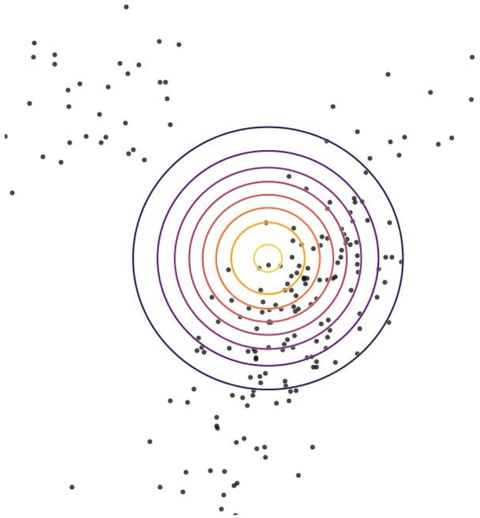
$$\downarrow$$

$$\theta^* = \arg\min_{\theta}\max_{\phi} \mathbb{E}_{x\sim p,\hat{x}\sim q_\theta} \mathrm{V}(f_\phi(x), f_\phi(\hat{x}))$$

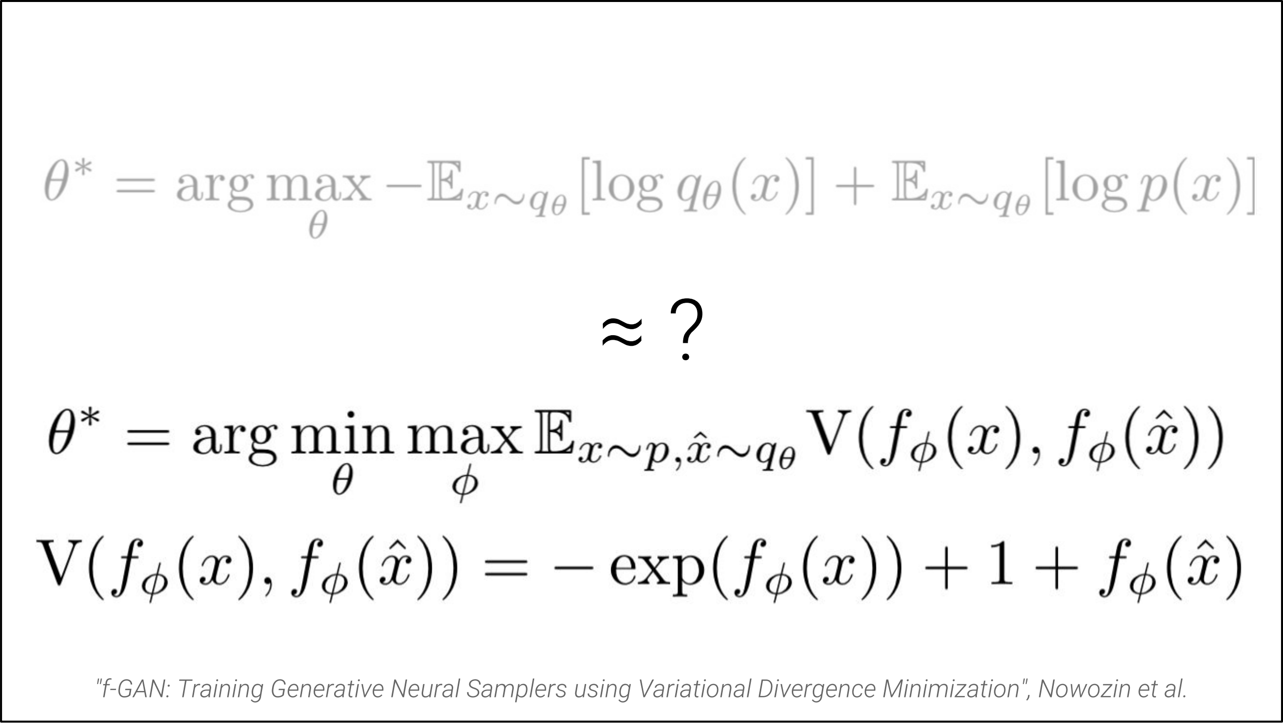
$$\mathrm{V}(f_\phi(x), f_\phi(\hat{x})) = -\exp(f_\phi(x)) + 1 + f_\phi(\hat{x})$$

"f-GAN: Training Generative Neural Samplers using Variational Divergence Minimization", Nowozin et al.

Here's an example of a loss function we might use for V. Importantly, using this particular loss function is equivalent in some far-off limit to optimizing the reverse KL divergence. Similarly, there are ways to effectively minimize other divergences like the Jenson-Shannon divergence or the earth mover's distance using the GAN framework. This opens up the possibilities of using divergences which would otherwise not be possible because we no longer have a term which measures the probability of a sample under the true distribution p(x). I want to emphasize the importance of this - maximum likelihood is an old and long-established technique, and in some sense was sort of the dogma in generative modeling for 100 years. This is not such a bad thing because it has some nice theoretical properties - for example, its efficiency and consistency - but GANs open up the possibility of training with something other than maximum likelihood, and this is huge.



Here's the result of learning the parameters of q theta using a GAN which corresponds, in the limit, to using the reverse KL divergence. You can see it learns a similar solution. Note again that when I showed the solution using the reverse KL I was cheating; here I don't have to cheat any more because there is no term which measures the probability of a sample under the true distribution.



$$\theta^* = \arg\max_{\theta} -\mathbb{E}_{x\sim q_\theta}[\log q_\theta(x)] + \mathbb{E}_{x\sim q_\theta}[\log p(x)]$$

$$\approx ?$$

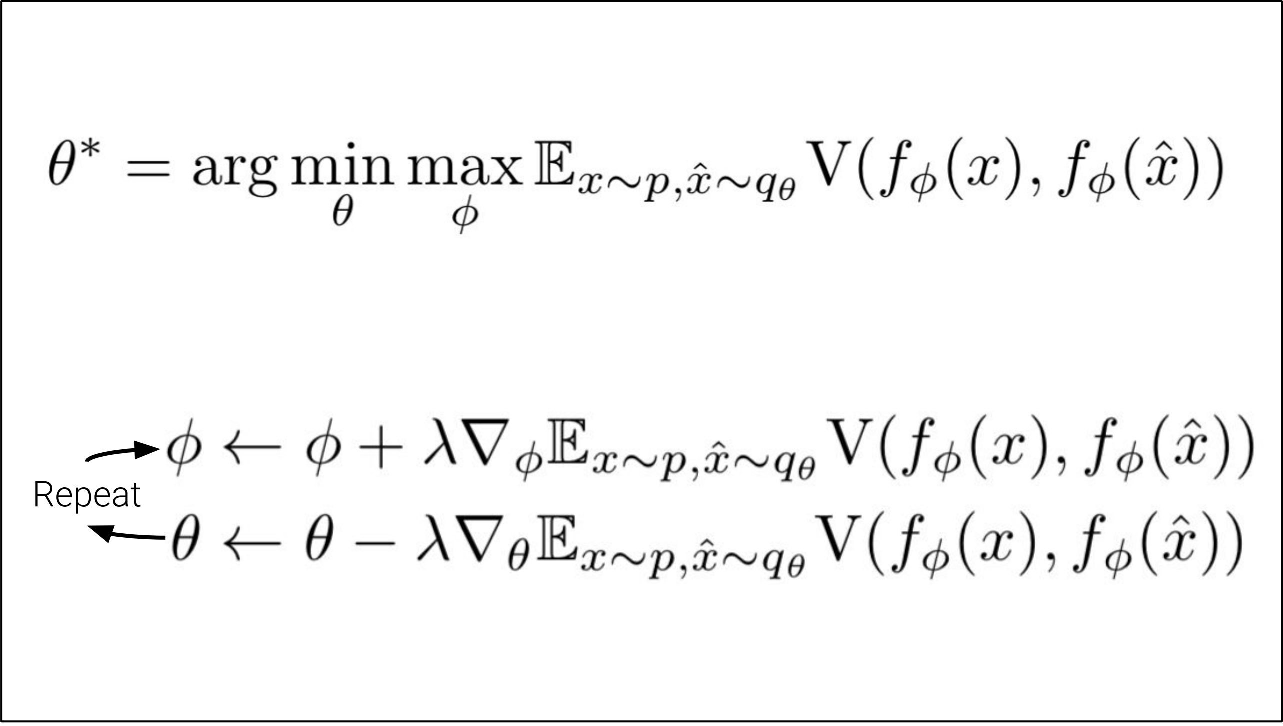
$$\theta^* = \arg\min_{\theta}\max_{\phi} \mathbb{E}_{x\sim p,\hat{x}\sim q_\theta} \mathrm{V}(f_\phi(x), f_\phi(\hat{x}))$$

$$\mathrm{V}(f_\phi(x), f_\phi(\hat{x})) = -\exp(f_\phi(x)) + 1 + f_\phi(\hat{x})$$

"f-GAN: Training Generative Neural Samplers using Variational Divergence Minimization", Nowozin et al.

Now, when we use a GAN we are not literally minimizing some underlying, well-defined divergence. Instead, we're optimizing some approximation of it, where the approximation comes from what f theta is and how we minimize it. In my opinion this actually makes GANs more interesting - for example, if f theta is a convolutional neural network, we incorporate a structural prior about shift invariance across certain dimensions; if f is a recurrent neural network, we incoporate a structural prior about dependencies across a temporal dimension, etc. This really is why I think it's useful to think of GANs as learning a loss function, where the form of the loss function is determined by the structure of the discriminator network.

I should mention at this point that everything I've discussed before is not new - but it's a perspective I really like and I think it often gets forgotten, in the sense that people just think of GANs as "those things that produce really nice images". From now on, I'm going to talk about some work with my colleagues which explores the GAN framework from some new angles.



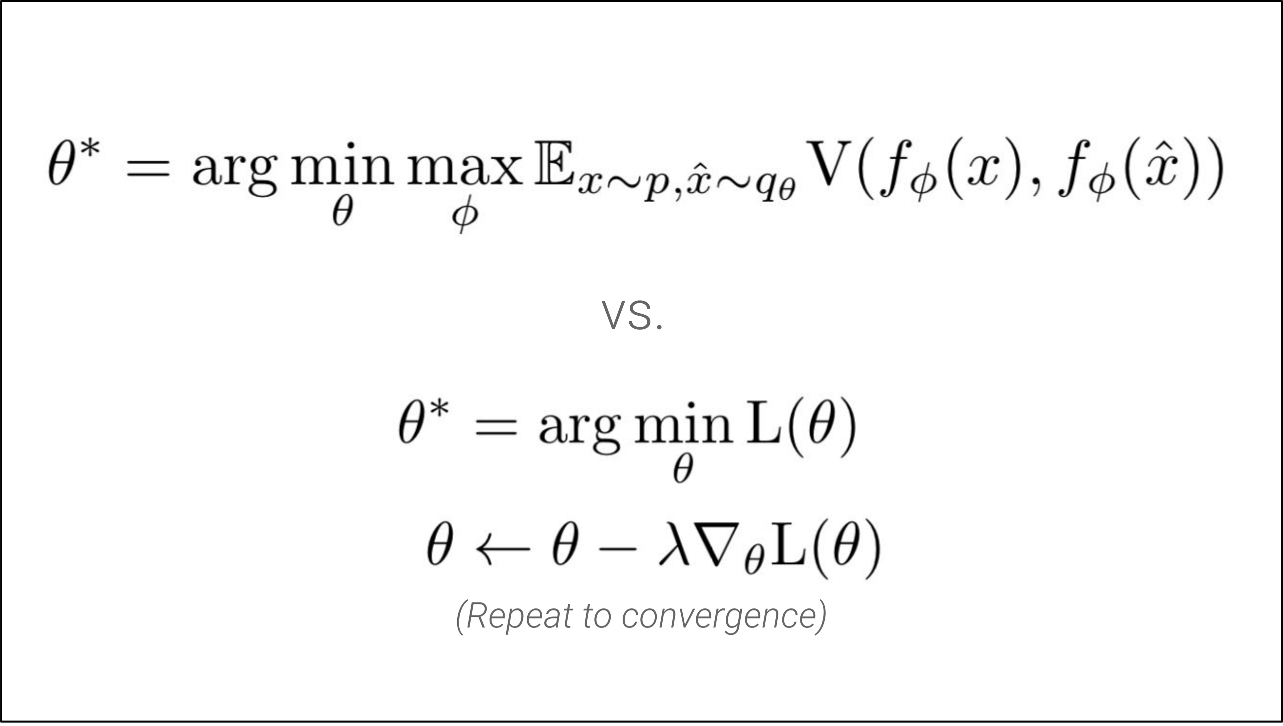
$$\theta^* = \arg\min_{\theta}\max_{\phi}\mathbb{E}_{x\sim p,\hat{x}\sim q_\theta}\mathrm{V}(f_\phi(x), f_\phi(\hat{x}))$$

$$\phi \leftarrow \phi + \lambda\nabla_{\phi}\mathbb{E}_{x\sim p,\hat{x}\sim q_\theta}\mathrm{V}(f_\phi(x), f_\phi(\hat{x}))$$

Repeat

$$\theta \leftarrow \theta - \lambda\nabla_{\theta}\mathbb{E}_{x\sim p,\hat{x}\sim q_\theta}\mathrm{V}(f_\phi(x), f_\phi(\hat{x}))$$

So, how do we optimize a GAN in practice? Note that we need to both maximize over phi and minimize over theta. In practice this is usually done by taking the gradient of the loss with respect to phi and taking a step in the positive gradient direction to update phi, and then taking the gradient with respect to theta and taking a step in the negative gradient direction. We use some small learning rate lambda for both steps. This is called "simultaneous gradient descent".



$$\theta^* = \arg\min_{\theta}\max_{\phi}\mathbb{E}_{x\sim p,\hat{x}\sim q_\theta}\mathrm{V}(f_\phi(x), f_\phi(\hat{x}))$$

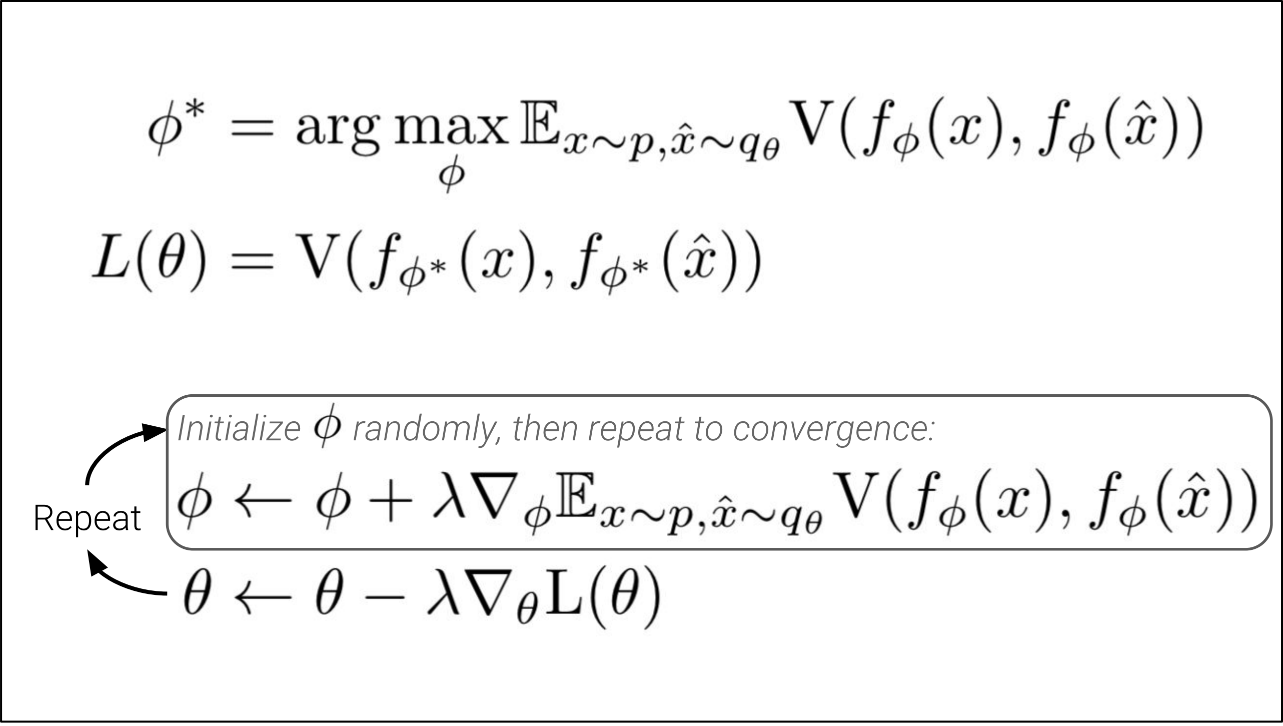
vs.

$$\theta^* = \arg\min_{\theta}\mathrm{L}(\theta)$$

$$\theta \leftarrow \theta - \lambda\nabla_\theta\mathrm{L}(\theta)$$

(Repeat to convergence)

This is actually pretty different from how we train normal machine learning models. Usually we have a consistent loss function, let's call it L(theta), and we want to find the value of theta which minimizes it. Then, we just do gradient descent to minimize it. In contrast, when we do simultaneous gradient descent, the loss function changes at each iteration.



$$\phi^* = \arg\max_{\phi} \mathbb{E}_{x \sim p, \hat{x} \sim q_\theta} \mathrm{V}(f_\phi(x), f_\phi(\hat{x}))$$

$$L(\theta) = \mathrm{V}(f_{\phi^*}(x), f_{\phi^*}(\hat{x}))$$

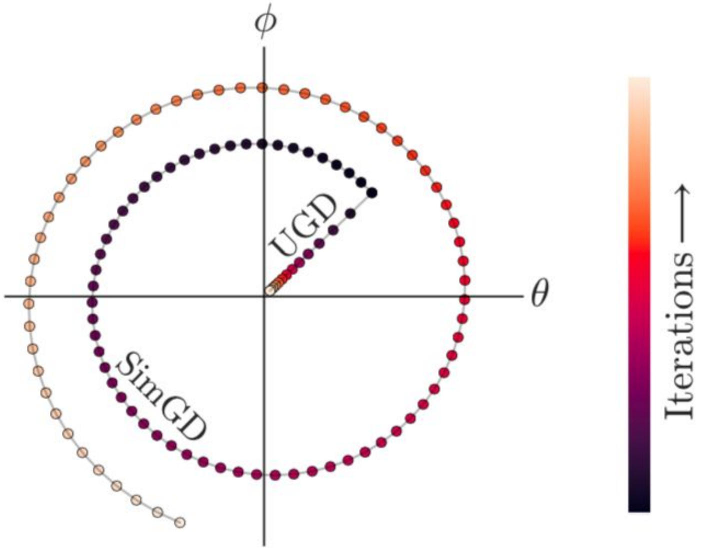
Initialize ϕ randomly, then repeat to convergence:

$$\phi \leftarrow \phi + \lambda \nabla_\phi \mathbb{E}_{x \sim p, \hat{x} \sim q_\theta} \mathrm{V}(f_\phi(x), f_\phi(\hat{x}))$$

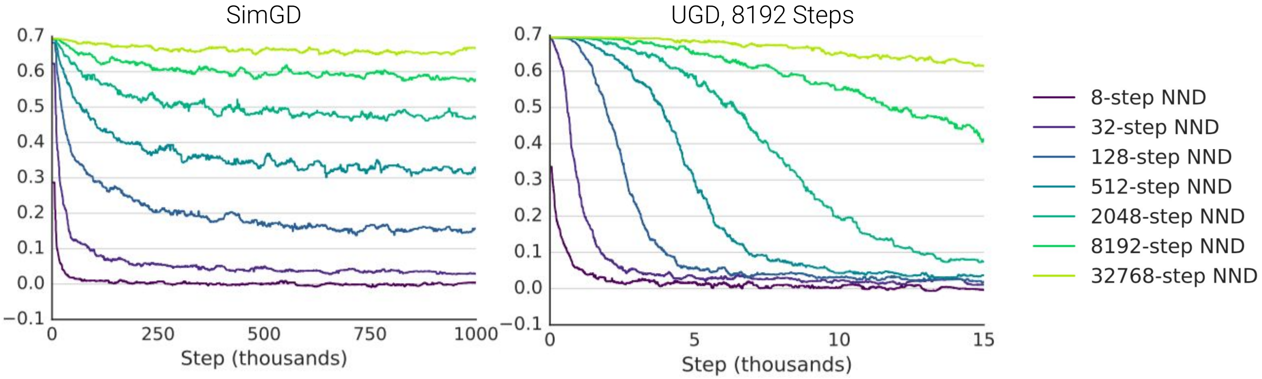
$$\theta \leftarrow \theta - \lambda \nabla_\theta \mathrm{L}(\theta)$$

But we can actually do this with a GAN. Let's define our loss function as optimizing, at every generator training step, the discriminator f to convergence, trained from scratch. Then, we can just do normal gradient descent on this loss function. We can "unroll" the optimization procedure used to train the critic, and optimize the generator through "unrolled gradient descent". Now, our loss function can be better considered a learned divergence. Usually, when people think of GANs, they think of simultaneous SGD by default. But that conflates the loss function and the optimization procedure. Here, we propose using a different optimization procedure, and ask how this changes GANs. Of course, it's totally impractical, because we have to train the critic from scratch to convergence at each generator update. So it's ridiculously expensive, and not a practical approach.

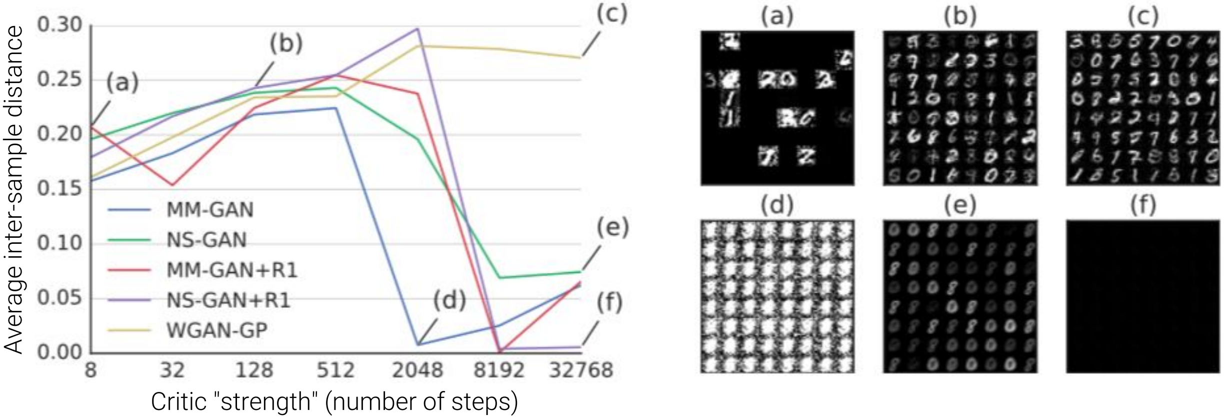
$$\min_{\theta} \max_{\phi} \theta\phi$$



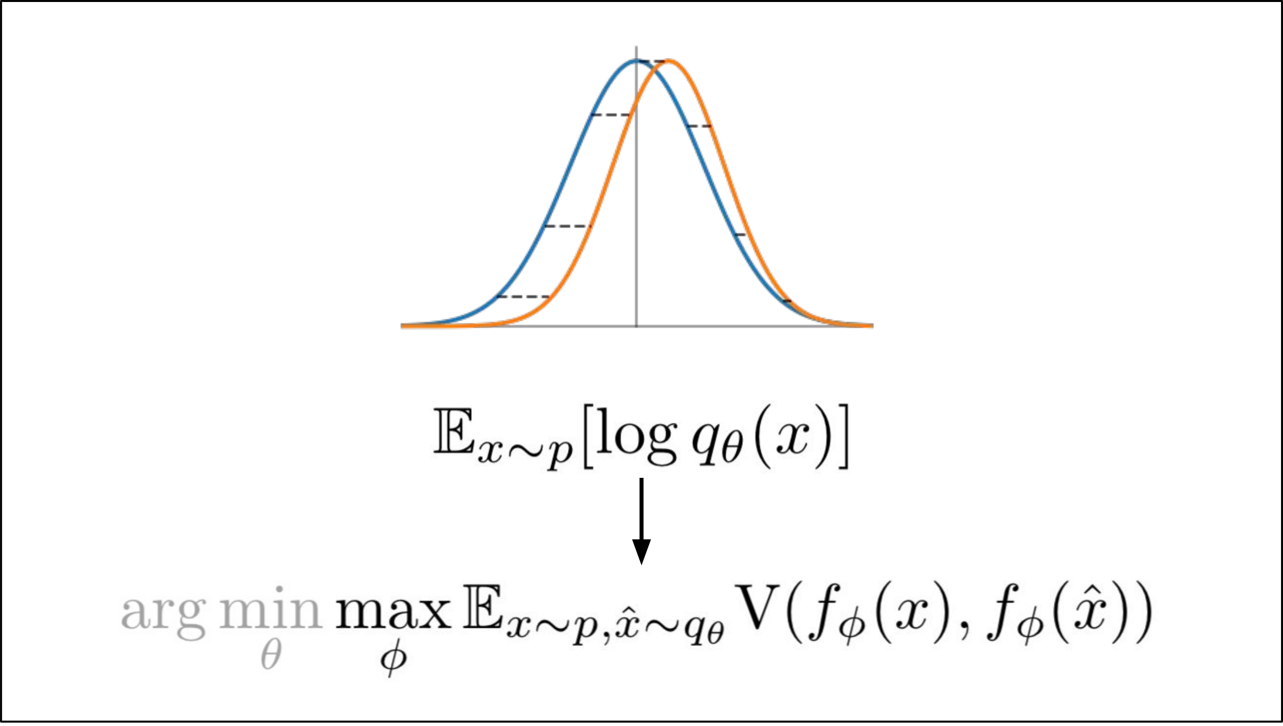
So, what happens when we do this? First of all, we can avoid some of the pathological behaviors commonly associated with GANs. A simple way people demonstrate one possible pathology with the GAN objective is by considering this simpler objective - theta times phi, where we want to minimize with respect to theta and maximize with respect to phi. This objective has an equilibrium point at 0, 0 - at this point we cannot increase or decrease the objective by changing theta or phi. SimGD actually never finds this solution because phi makes the objective worse for theta, and then theta makes the objective worse for phi, and so on… unrolled gradient descent does find this solution because optimization of phi resets every time we update theta.



Another thing we can check is that minimizing this learned divergence with unrolled gradient descent does better than minimizing it with simultaneous gradient descent. We can see here that when we train a GAN with simultaneous gradient descent, it does a reasonable job of minimizing a learned divergence trained for 8 steps. However, even if we train it for a million steps, it never does a very good job fooling a critic which has been trained for, say, a few thousand steps. In contrast, if we train the same GAN with unrolled gradient descent, unrolling for 8000 steps, we can see it is doing a reasonable job minimizing this learned divergence even after only 15,000 steps.



Now that we have another way of optimizing the GAN objective, we can ask questions like "should common behaviors of GANs be attributed to the loss functions used or the optimization procedure?" For example, one common problem with GANs is "mode collapse", where the generator just repeatedly generates a single (possible garbage) output and makes no further progress during training. Is mode collapse caused by simultaneous gradient descent or a poor loss function? If we train different GANs using different loss functions with unrolled gradient descent, with a different amount of unrolling, we see that some GAN functions exhibit mode collapse when trained against a "stronger" critic (one which has been trained longer). There is a loss which did not exhibit mode collapse - the WGAN-GP, which managed to avoid mode collapse even when trained against a critic which has been unrolled for 30,000 steps.



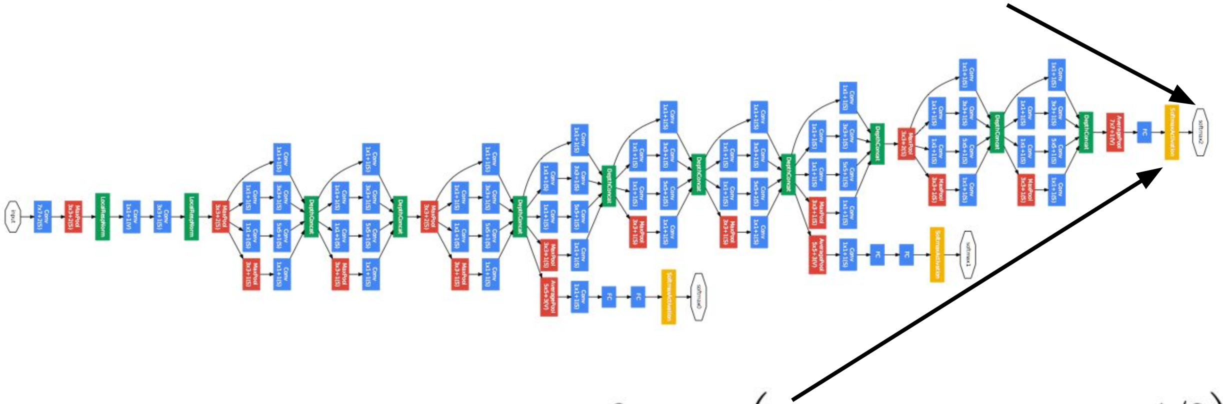
$$\mathbb{E}_{x\sim p}[\log q_\theta(x)]$$

$$\big\downarrow$$

$$\arg\min_{\theta}\max_{\phi}\mathbb{E}_{x\sim p,\hat{x}\sim q_\theta}\mathrm{V}(f_\phi(x),f_\phi(\hat{x}))$$

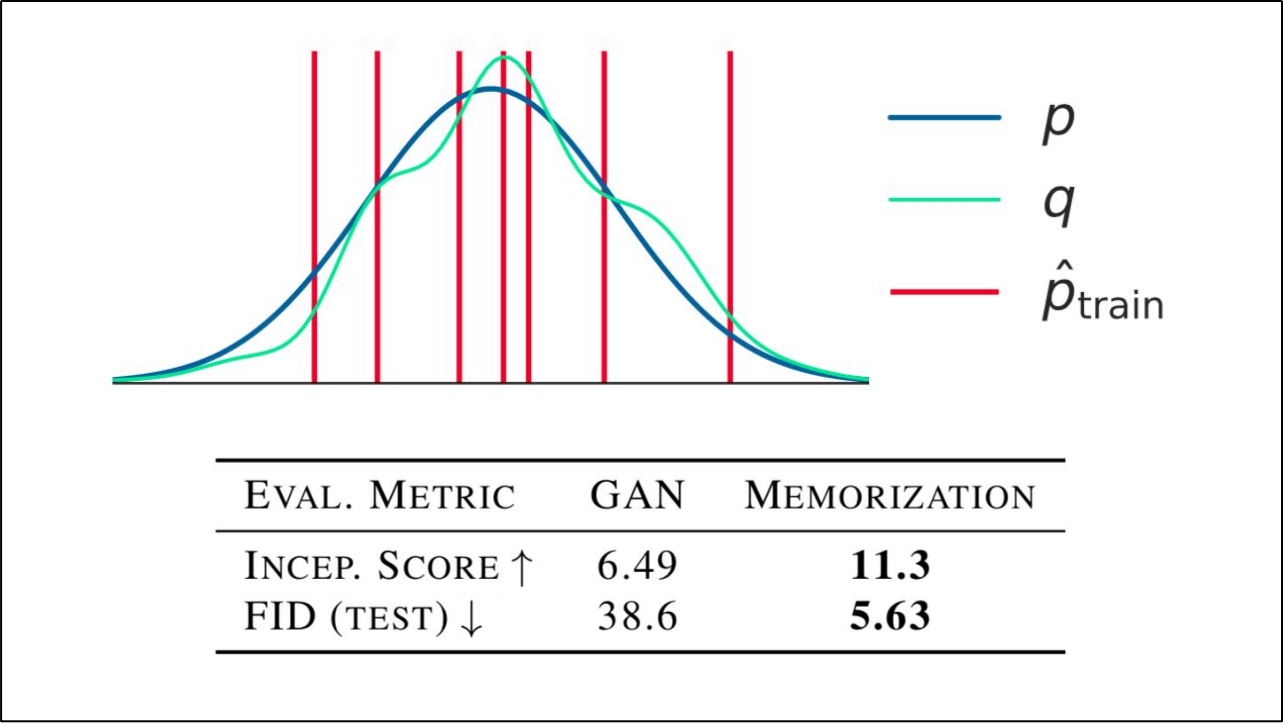
So, we have been talking GANs as minimizing a learned divergence, and exploring different ways of minimizing it. Let's make things a little simpler. Measuring a divergence is a good way to evaluate how similar two distributions are - for example, if we want to measure how closely our model matches the true distribution. A common way to do that is to measure the likelihood of samples from the true distribution under the estimated distribution - remember that this is roughly equivalent to measuring the KL divergence. However, now we have a new way to measure a rich class divergences - we can train a critic network by optimizing to some loss function, and then use the value of the loss function after training as a measure of some divergence. Why might we want to do this? Well, remember that measuring likelihood requires being able to evaluate the probability of a sample under our estimated distribution. There are some generative models, for example GANs, which don't allow us to compute this - we can only draw samples from them.

$$\mathrm{IS}(q) = \mathbb{E}_{x\sim q}[D_{\mathrm{KL}}(p(y|x)\|p(y))]$$



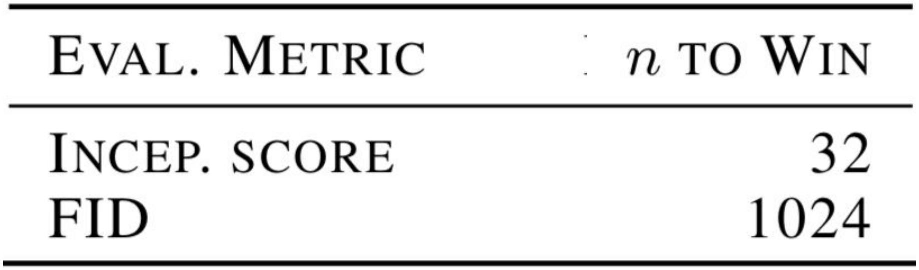
$$D_{\mathrm{FID}}(p,q) = \|\mu_p - \mu_q\|^2 + \mathrm{Tr}\left(\Sigma_p + \Sigma_q - 2(\Sigma_p\Sigma_q)^{1/2}\right)$$

There are actually some evaluation methods which you may already be familiar with which are designed to measure the similarity between two distributions based only on samples - the Inception Score and the Frechet Inception Distance. These are designed for comparing distributions of natural images, and are computed by feeding samples from each distribution into an Inception network. The two scores are computed by computing statistics about the real and generated images. They've both been shown to correlate with human judgement of perceptual quality.



EVAL. METRIC	GAN	MEMORIZATION
INCEP. SCORE ↑	6.49	**11.3**
FID (TEST) ↓	38.6	**5.63**

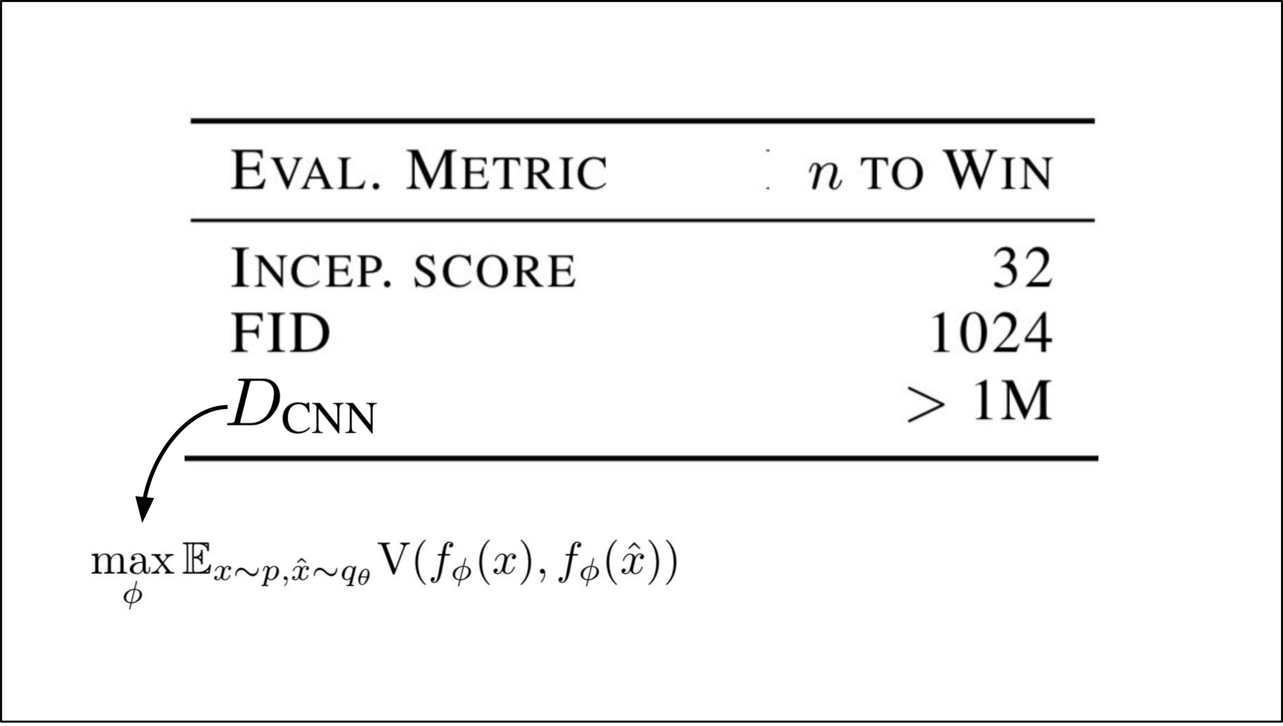
While these approaches are widely used, they have a pretty odd characteristic - namely, that they prefer training set memorization. What I mean by this is that the score achieved by the training set is actually better than what is achieved by a good generative model. For most downstream tasks, this is not a useful solution. Another way of putting this is that they prefer training set memorization (shown in red) to a model (shown in green) which imperfectly fits the true distribution (p, blue) but covers more of p’s support.



EVAL. METRIC	n TO WIN
INCEP. SCORE	32
FID	1024

How many training set images n does one need to memorize to score better than a well-trained GAN model?

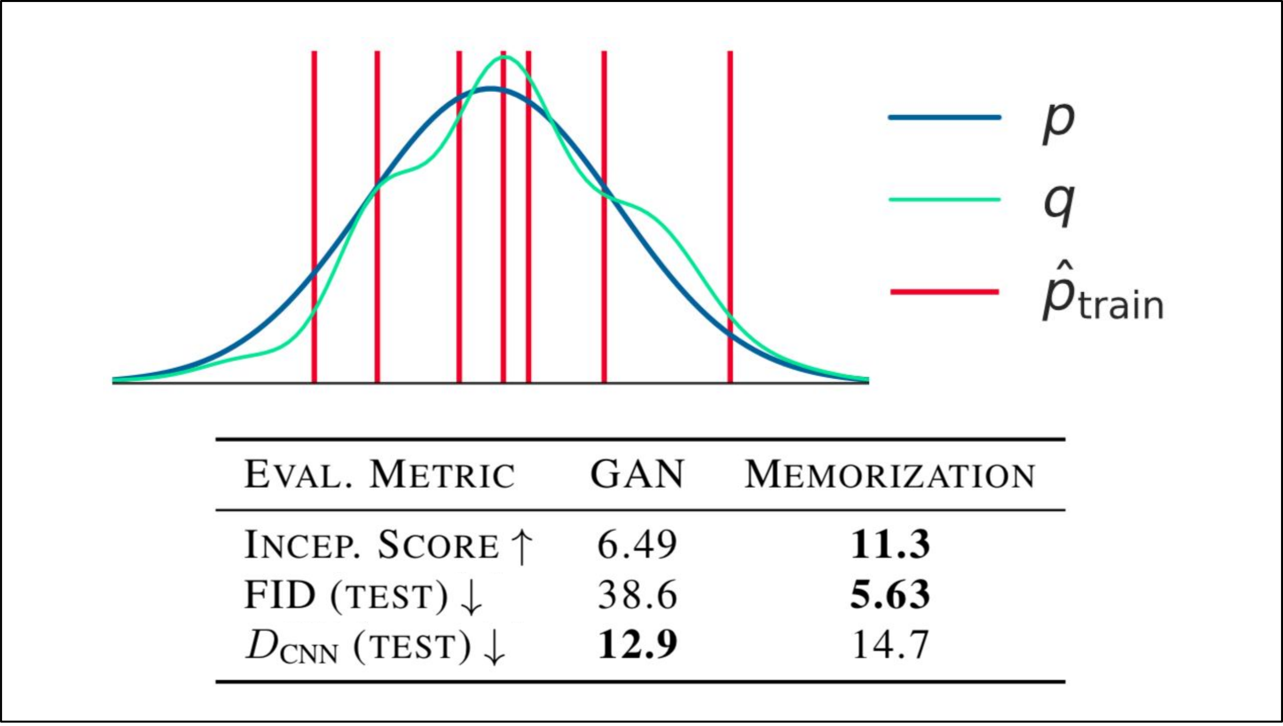
A simple way to measure this is to train a decent generative model (here, a GAN) and measure the score achieved by samples from the model. Then, measure the score achieved by memorizing differing amounts of the training set. We can measure, in a loose sense, how sensitive a score is to diversity by measuring the point at which the score prefers a memorized set of *n* images to the samples from the GAN. if *n* is small, it means that it prefers a small but highly "realistic" sample to a potentially huge but less realistic sample.



EVAL. METRIC	n TO WIN
INCEP. SCORE	32
FID	1024
D_{CNN}	$> 1\text{M}$

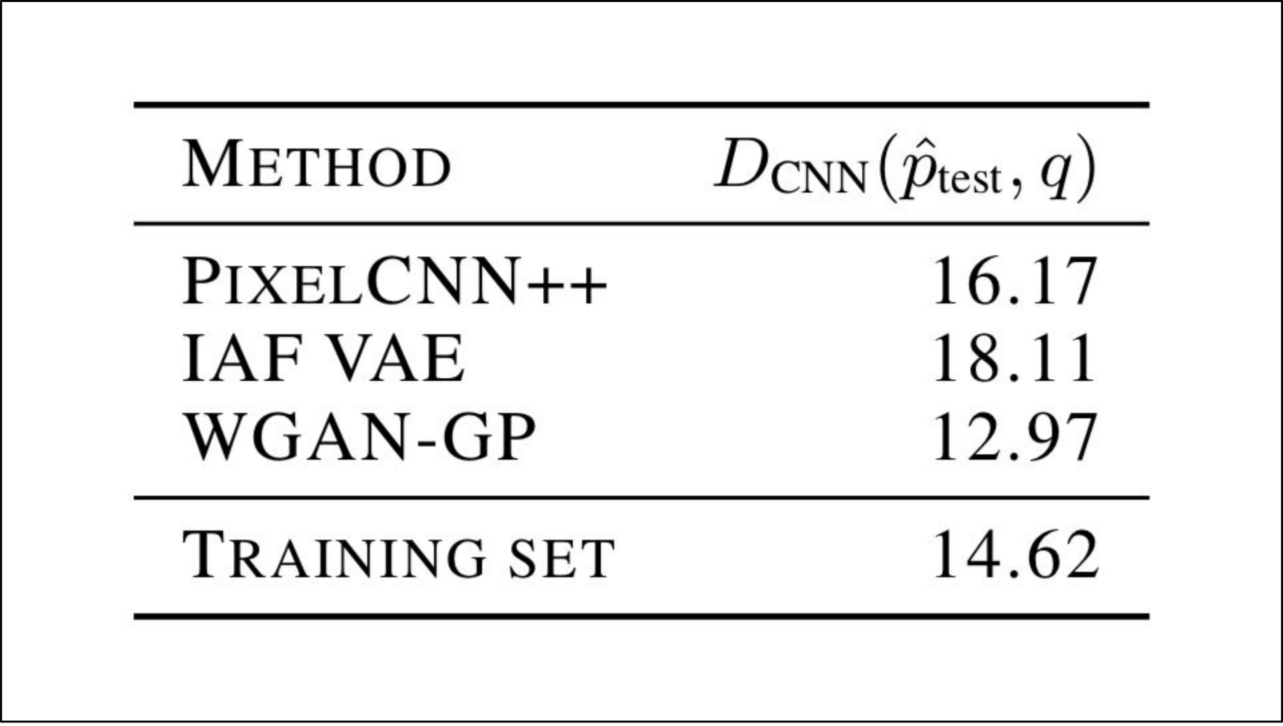
$$\max_{\phi} \mathbb{E}_{x \sim p, \hat{x} \sim q_\theta} \mathrm{V}(f_\phi(x), f_\phi(\hat{x}))$$

In our work, we investigate the use of GAN-based divergences for evaluation. If we train a critic network from scratch on samples from a pre-trained generative model and samples from the true distribution, and we measure its loss value at convergence, we also get a score. We call this score D_cnn, because in this case it's a divergence defined by a convolutional neural network critic. We can similarly measure the the point at which D_CNN prefers a memorized set of n images to the samples from the GAN. In this particular experiment, our critic metric needs a million images to prefer training set memorization to the samples produced by a GAN. In other words, it's much more sensitive to diversity than it is to quality.



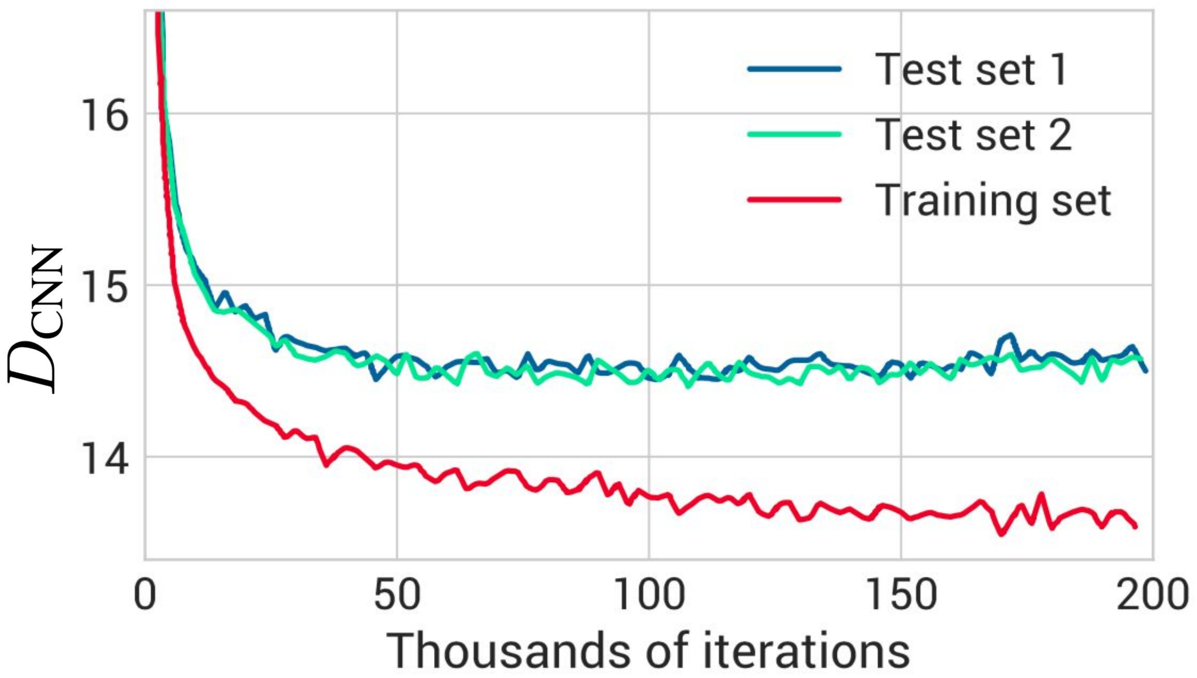
Eval. Metric	GAN	Memorization
Incep. Score ↑	6.49	**11.3**
FID (test) ↓	38.6	**5.63**
D_{CNN} (test) ↓	**12.9**	14.7

Now, if we compare the "memorization" solution with a GAN with D CNN, we see that it actually prefers the GAN. The main takeaway here is that this particular score prefers a more diverse, but ostensibly lower-quality generative model to simply memorizing the training set, which is high-quality but not particularly diverse - it can only generate on the order of ten thousand distinct samples..

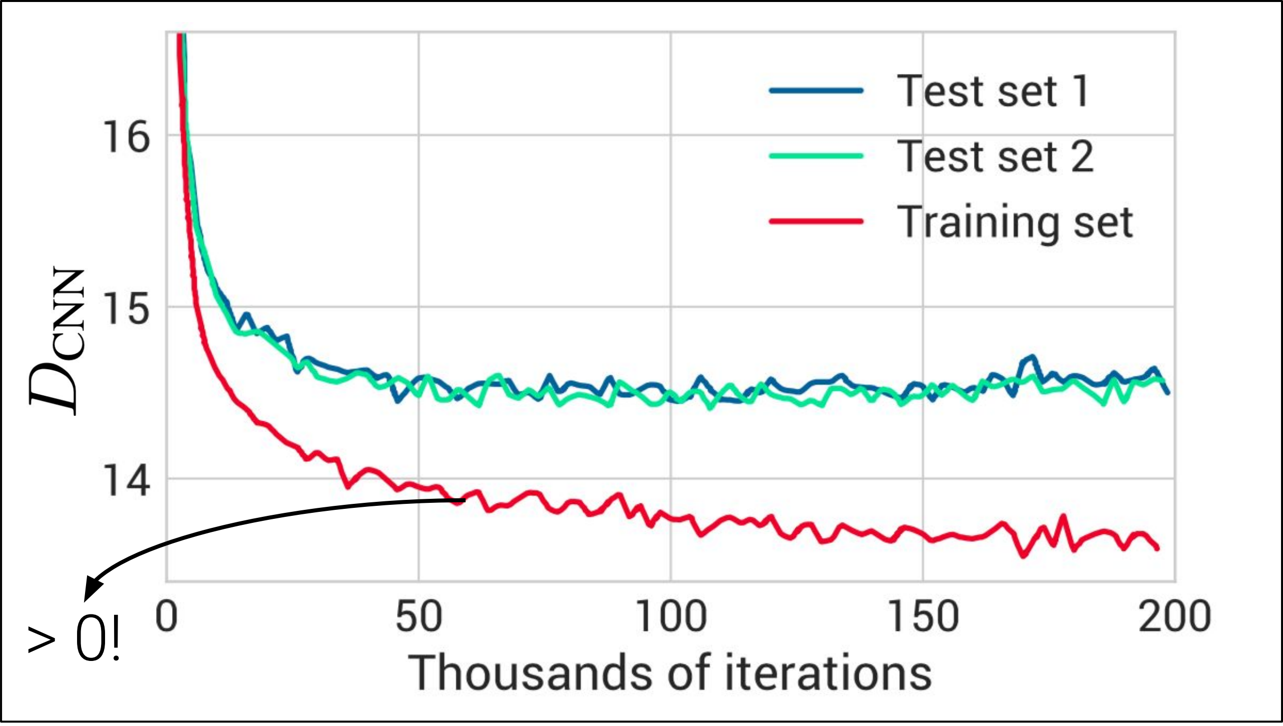


METHOD	$D_{\text{CNN}}(\hat{p}_{\text{test}}, q)$
PIXELCNN++	16.17
IAF VAE	18.11
WGAN-GP	12.97
TRAINING SET	14.62

Note that D_CNN can be used to evaluate any generative model, so we can compute this score for other types of generative models beyond GANs. This allows us to compare different classes of models on equal footing. In this case, a GAN outperforms a PixelCNN and a VAE model. Note that this does not imply the GAN is better in an absolute sense - just that the characteristics "preferred" by D CNN are better satisfied by the GAN than the other models. We can also compute this score on the training set, and the fact that the resulting score is higher for the training set than for the GAN suggests that the GAN has "generalized" in some sense which is preferred by this metric.

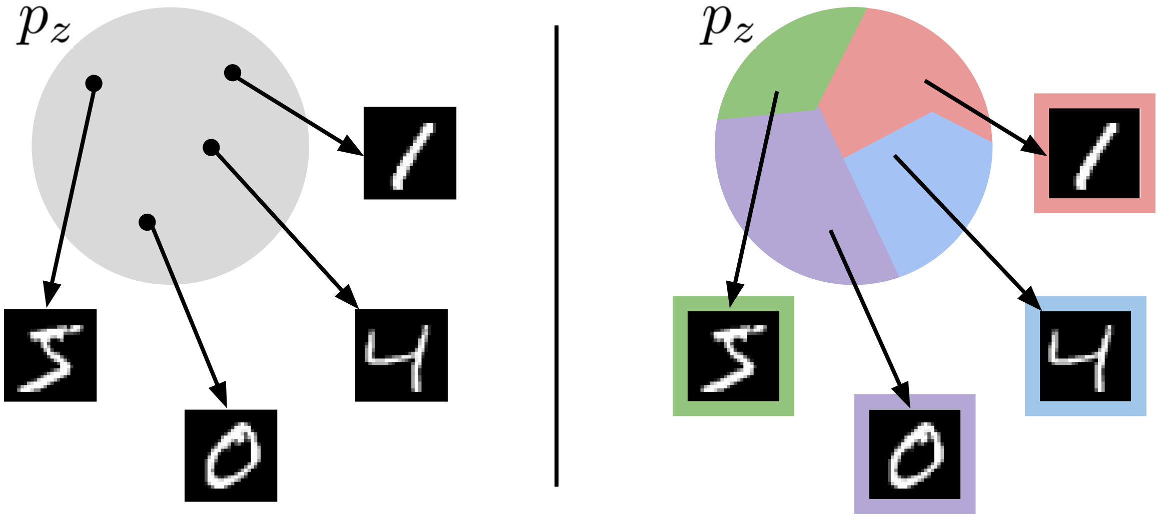


The fact that D CNN can prefer a sufficiently diverse and high-quality generative model to training set memorization allows us to do things like measure overfitting! We do so by measuring D CNN between samples from the generative model and samples from either the training set or the test set. Note that D CNN between the training set and itself would be zero, so the "gap" between D CNN on the training set and on the test set gives us some notion of overfitting. We see that over the course of training this particular model, which is another GAN, there is some degree of overfitting.

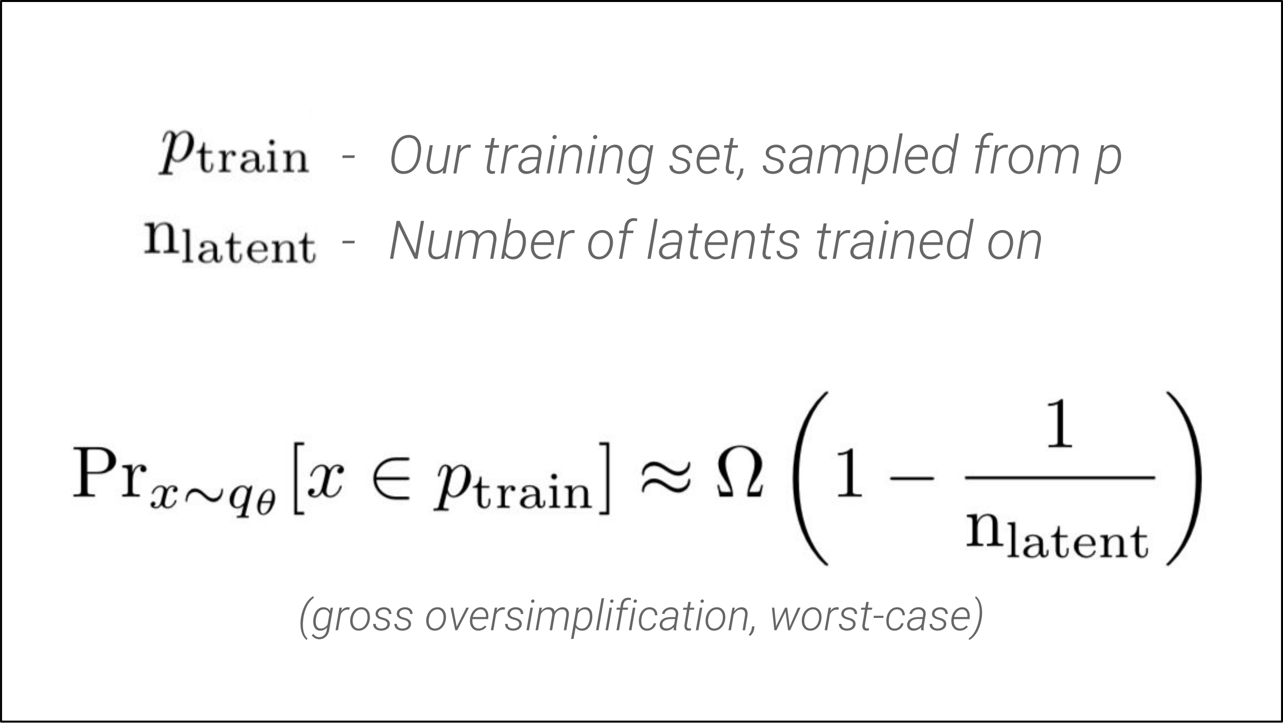


An important thing to note here is that D CNN on the training set never actually gets that close to zero. Of course, it's not clear how meaningful the difference between 14 and 0 is; we just know that it's lower-bounded by zero. This suggests that this particular GAN may have learned a generative model without just memorizing the training set. This is a pretty well-known property of GANs - for a long time people have been pretty thorough about finding nearest neighbors in the training set to samples from a GAN.

$$q_\theta(x) = G(z), z \sim p_z$$



In some parallel work, we've started to investigate this theoretically. To set up this idea, let's consider the following question: To what extent has a GAN memorized the training set? For some additional background if you're not familiar, the generative model learned by a GAN maps samples from a simple prior distribution p_z over a "latent space" to samples in the "data space" by passing it through a deterministic "generator" network, written as G(z) at the top. As we train the model, we sample latent vectors and feed them through the generator to create samples from q_theta. Now, we can consider two possibilities. The first, on the left, is that for a few distinct points in the latent space to map to training point examples. In this scenario, the training set basically has measure zero in the latent space, and the rest of the latent space maps to samples which are not in the training set. The second scenario, on the right, is where entire regions of the latent space have been mapped to training datapoints. In this case, no matter where we sample in latent space, the generator will end up producing one of these four images from the training set.



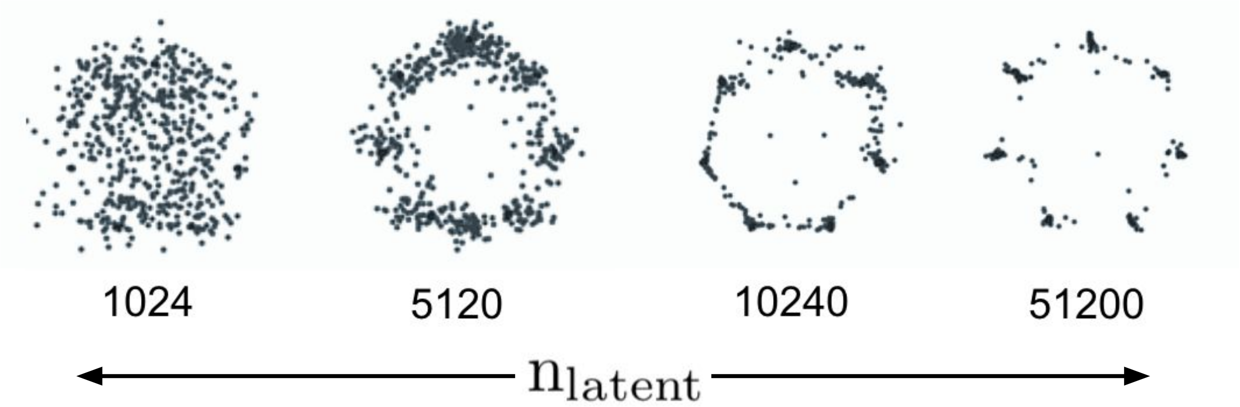
p_{train} - *Our training set, sampled from p*

$\mathrm{n}_{\text{latent}}$ - *Number of latents trained on*

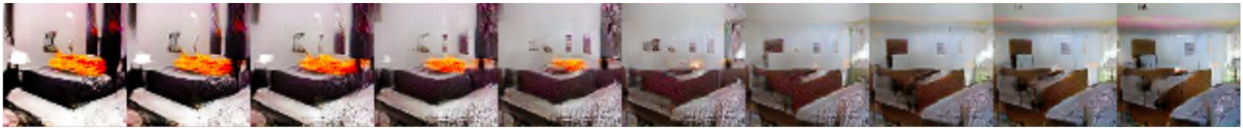
$$\Pr_{x \sim q_\theta}[x \in p_{\text{train}}] \approx \Omega\left(1 - \frac{1}{\mathrm{n}_{\text{latent}}}\right)$$

(gross oversimplification, worst-case)

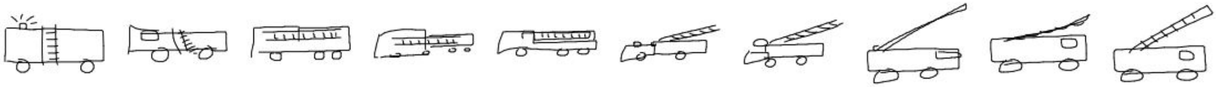
Now, in this work, we study the following question: What relationship does the number of latent vectors the generator has been trained on have to the extent to which it has memorized the training set across the entire latent space? If we denote p_train as the sample from the true distribution that we train the GAN on, and n_latent as the number of latent vectors that the GAN has been trained with, then we show (under some strong assumptions) something which looks like what's shown on the bottom: Namely, the probability that a sample from the GAN is in the training set is proportional to 1 minus 1 over the number of latent vectors. In other words, as we train our GAN on more latent vectors, the GAN is more likely to have memorized the training set. Note that this is a gross approximation of a worst-case bound, but it communicates the basic idea of the theorem. As a note, typically n_latent is the number of training steps times the batch size - in other words, at each training step we sample a new batch of latent vectors.



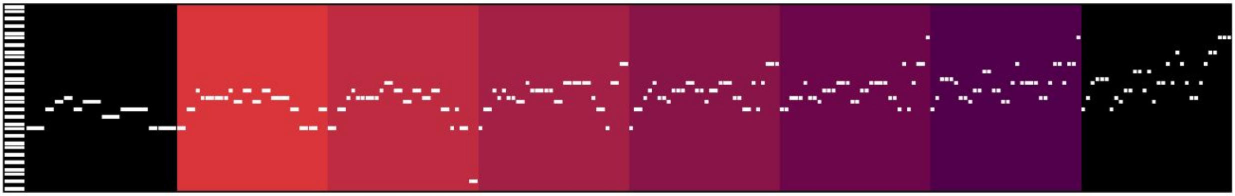
Here's a simple illustration of this on a toy dataset, consisting of 7 points around a circle. We train a GAN on a *fixed* set of latent vectors - before training, we sample n_latent vectors, and then at each iteration of training the generator we sample a minibatch of latent vectors from this fixed set. Shown in each of these plots is the result of training a GAN with varying numbers of fixed latent vectors. The plots show generated samples for latent vectors sampled at random. not just those used to train the model. As expected, as the number of fixed latent vectors grows, the probability that a sample from the model is similar to a training datapoint also approaches 1. Now, on real datasets, what we've found empirically is that the number of latent vectors required for training set memorization is way larger in practice than what is typically used to train a GAN, but more investigation is needed.



Radford et al., "*Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*"

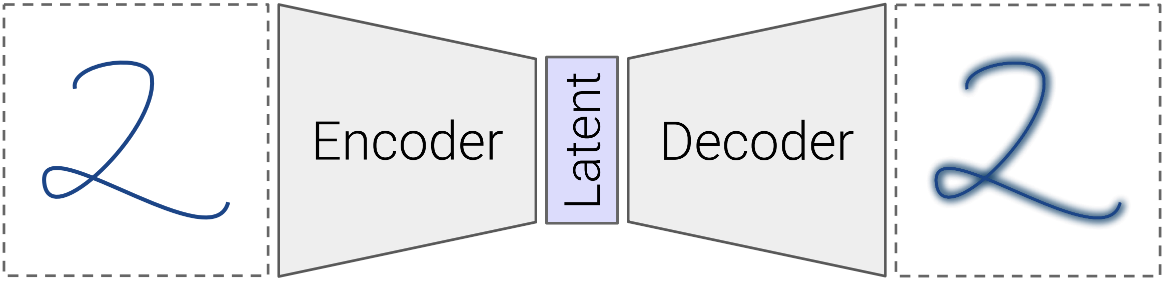


Ha & Eck, "*A Neural Representation of Sketch Drawings*"

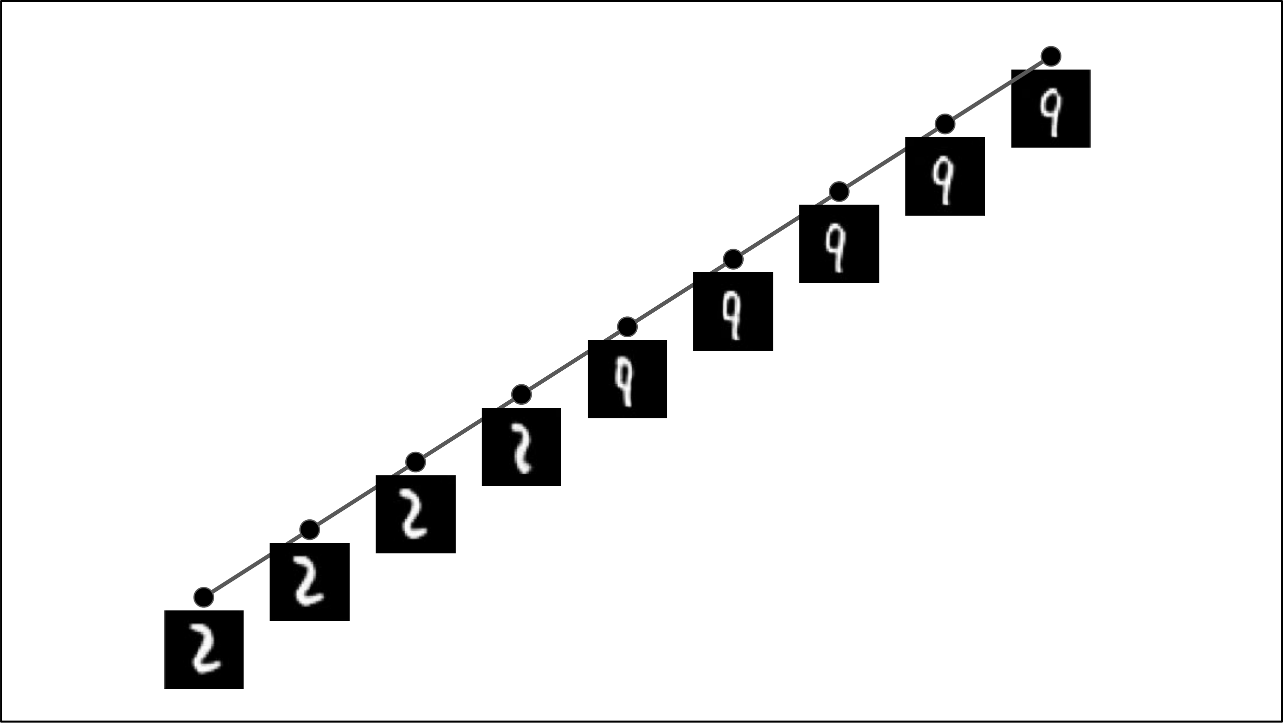


Roberts et al., "*A Hierarchical Latent Vector Model for Learning Long-Term Structure in Music*"

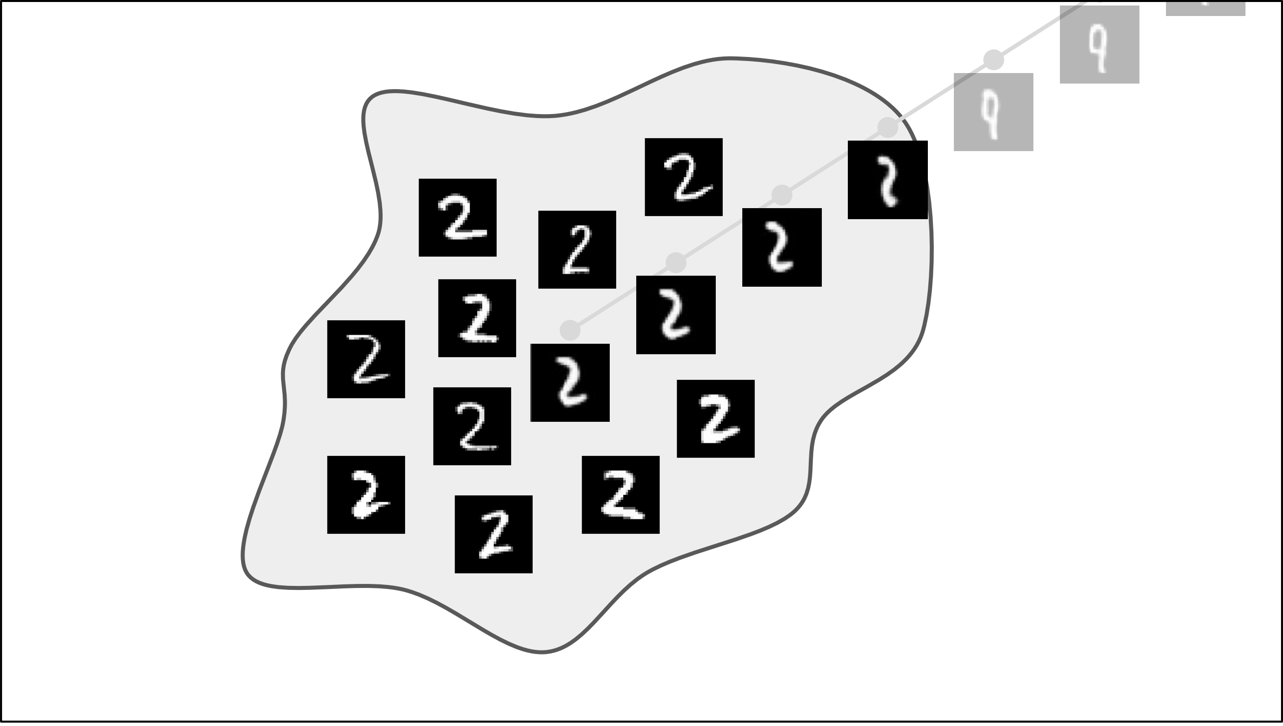
So what does it mean if the model has not "overfit" to the training set, and has produced a continuous latent space? One thing this kind of model can often do is interpolate. Here we see interpolations generated by the DCGAN, sketch-RNN, and our MusicVAE model. The first model is a kind of GAN model; the other two are autoencoders, which I'll describe in a moment. For this last part of my talk, I'll turn to a broader question: Does the ability to interpolate suggest that the autoencoder has learned a useful representation of this data?



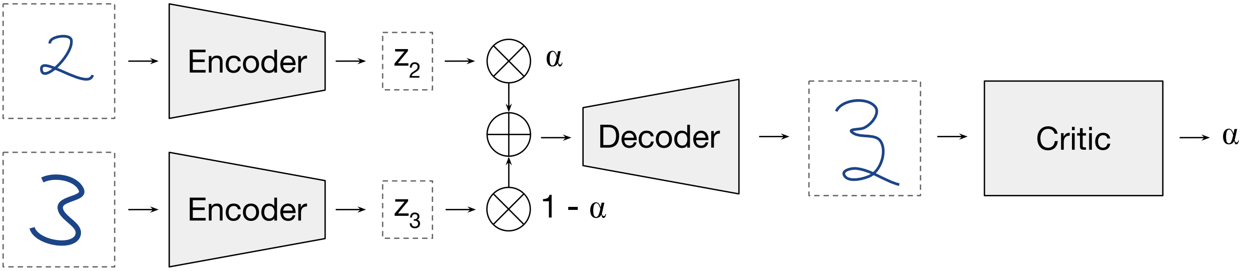
The autoencoder is a simple model structure whose goal is to reconstruct its input. The input is first mapped to a latent code, which is typically lower-dimensional than the input. The encoder tries to encode all of the useful information about the input into into the latent code, and the decoder tries to reconstruct the input based only on the information in the latent. The model is typically trained against a reconstruction error cost such as mean squared error, which measures how closely the reconstruction matches the input.



When we interpolate between two points in latent space, we're traversing the line in latent space between the encoding of point A and point B and decoding the result along the line. We say an interpolation is good if it semantically smoothly varies from point A to point B, and if it remains realistic across the interpolation. The first point requires that the latent space is "smooth", the second requires that it has no "holes" between data points where data becomes unrealistic.



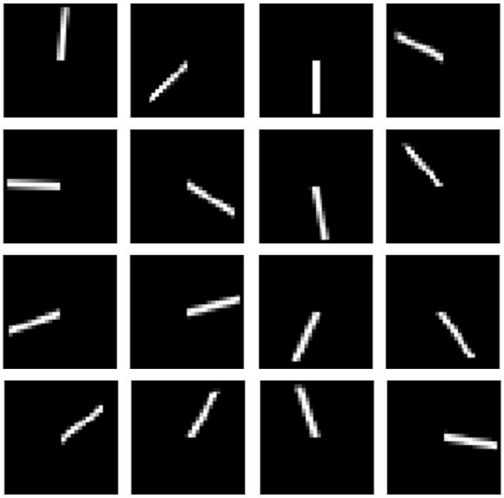
So what does it mean that the latent space is smooth? It means that nearby each latent code are the latent codes corresponding to semantically similar datapoints. By extension this suggests some structure in the latent space - namely that the latent space is in some loose way organized around semantic similarity. So in this example, all of the similar twos are clustered near each other, because if we move a small amount away from one of the twos we should get a two-like symbol. This suggests that there may be some kind of link between a latent space which allows interpolation and whether it could be used as a high-quality learned representation for downstream tasks.



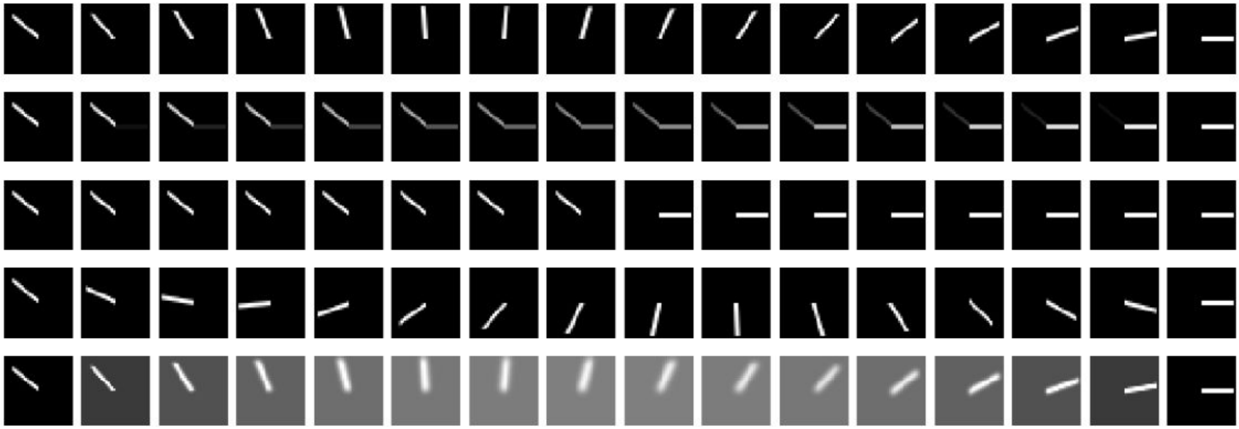
Adversarial **C**onstraint for **A**utoencoder **I**nterpolation

(Disclaimer: This is not a GAN)

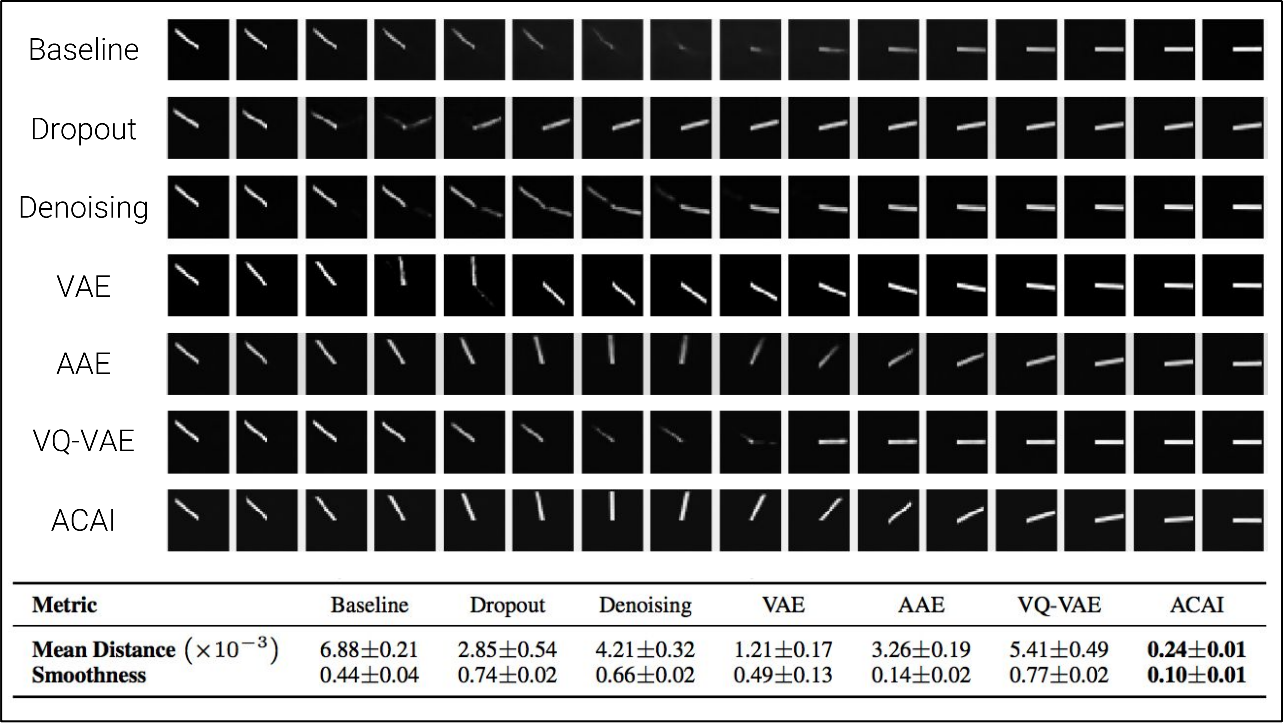
But now I'm going to talk about a more explicit way of encouraging good interpolations. This will allow us to test, in isolation, whether encouraging high-quality interpolations also produces useful representations. What is our goal when we interpolate? As I mentioned, one goal is that intermediate points are realistic, or in other words, are indistinguishable from "real" datapoints. We propose a regularizer for autoencoders which explicitly enforces this goal. We take two datapoints, encode them, interpolate their latent codes with some mixing coefficient alpha, then decode the result. Then, we train a critic to try to distinguish between reconstructions of real datapoints, and reconstructions of interpolated latent codes. It does this by trying to predict the mixing coefficient alpha. The autoencoder, in turn, is trained to force the critic to output alpha = 0 for interpolated points, which would correspond to not mixing at all - in other words, producing real datapoints. The critic implicitly learns to distinguish between the distribution of interpolated reconstructions and real reconstructions, which gives the autoencoder a useful objective to use to make its interpolations more realistic. We call this approach "adversarial constraint for autoencoder interpolation", or ACAI. Note that this is not a GAN, nor is it a generative model. But it shows another reasons the GAN framework is useful - it allows us to learn and optimize a divergence between totally arbitrary objects -- in this case, reconstructions and interpolants.



To test this idea, we'll start with a toy task where we are autoencoding black-and-white images of lines. The lines are radii of a circle inscribed in the border of the image. In this toy setting, we know the underlying manifold of the data - it's one dimensional, corresponding to the angle of the line.

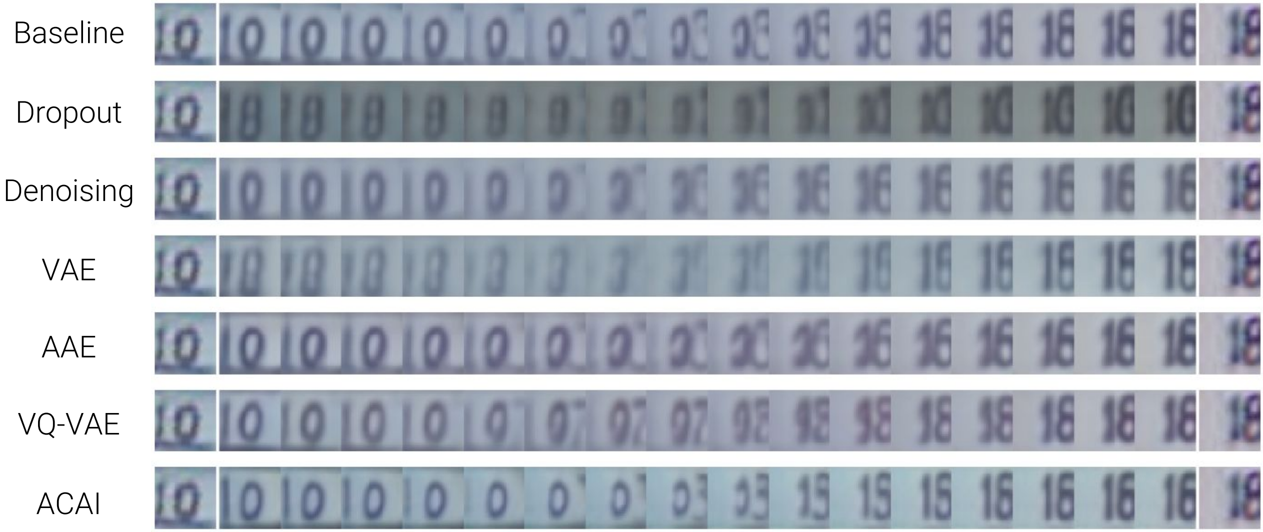


Here are some made-up examples of what good and bad interpolation on this dataset could look like. In this simple toy setting, we know what interpolating linearly on the data manifold should look like, and that's shown on the top. We are simply adjusting the angle of the line from the left endpoint to the right endpoint. The intermediate points look realistic. In the second row, we are interpolating in "data space", not along the data manifold. In the third row, we're abruptly moving along the data manifold, rather than smoothly. In the fourth row, we're interpolating smoothly but are not taking the shortest path along the manifold. In the final row, we are interpolating correctly but the intermediate points don't appear realistic.

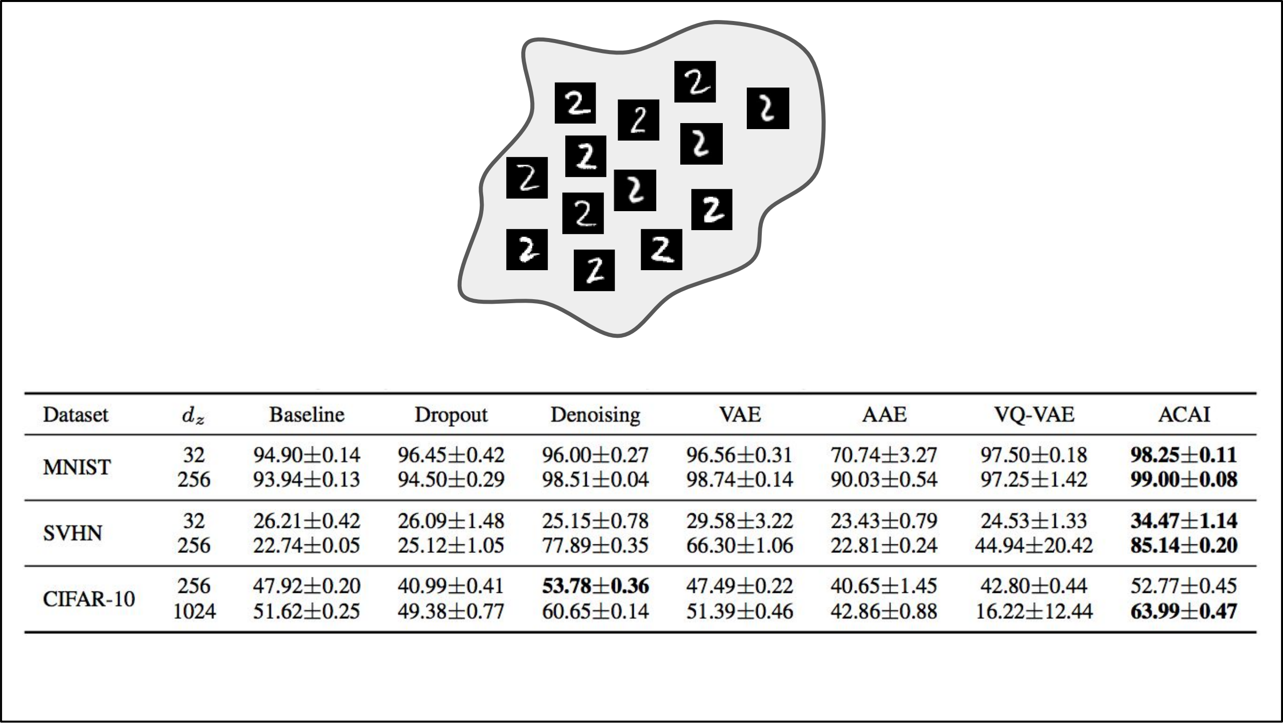


Metric	Baseline	Dropout	Denoising	VAE	AAE	VQ-VAE	ACAI
Mean Distance $(\times 10^{-3})$	6.88±0.21	2.85±0.54	4.21±0.32	1.21±0.17	3.26±0.19	5.41±0.49	**0.24±0.01**
Smoothness	0.44±0.04	0.74±0.02	0.66±0.02	0.49±0.13	0.14±0.02	0.77±0.02	**0.10±0.01**

So, how do common autoencoders fare on this task? Here we show a whole slew of them, including our proposed regularizer. If we just train a normal autoencoder with no constraint on the latent code, the intermediate points stop looking realistic when we interpolate. If we apply dropout to the latent code, we see "abrupt" interpolation behavior. The denoising autoencoder does data-space interpolation. Surprisingly, the VAE exhibits abrupt interpolation behavior. The adversarial autoencoder seems to interpolate reasonably well but the intermediate lines stop looking as realistic. The Vector-quantized autoencoder stops appearing realistic, like the baseline. Finally, ACAI applied to the baseline, with no other constraints, interpolates exactly as we'd hope. We proposed some simple heuristic scores to measure interpolation quality and found that ACAI performed best.

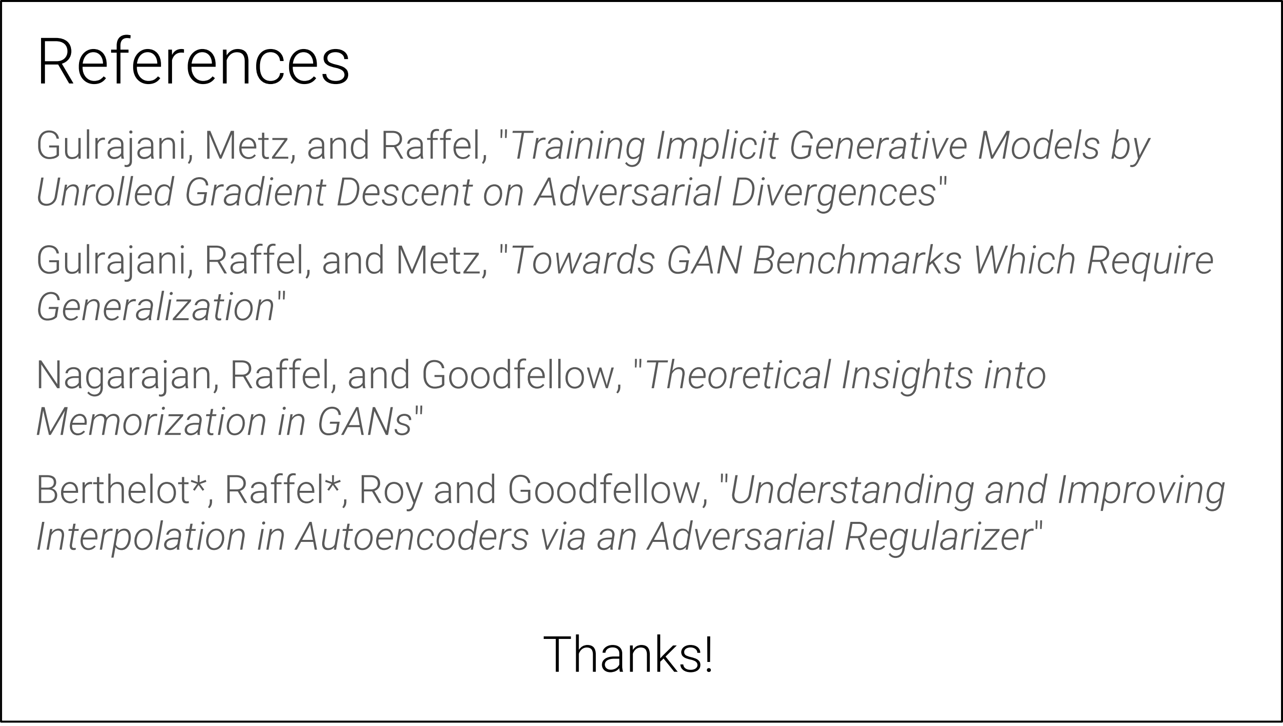


ACAI also interpolates well on real data. Other autoencoders are a bit more blurry, or intermediate datapoints are not realistic like on the baseline, but really they all do reasonably well.



Dataset	d_z	Baseline	Dropout	Denoising	VAE	AAE	VQ-VAE	ACAI
MNIST	32	94.90±0.14	96.45±0.42	96.00±0.27	96.56±0.31	70.74±3.27	97.50±0.18	**98.25±0.11**
	256	93.94±0.13	94.50±0.29	98.51±0.04	98.74±0.14	90.03±0.54	97.25±1.42	**99.00±0.08**
SVHN	32	26.21±0.42	26.09±1.48	25.15±0.78	29.58±3.22	23.43±0.79	24.53±1.33	**34.47±1.14**
	256	22.74±0.05	25.12±1.05	77.89±0.35	66.30±1.06	22.81±0.24	44.94±20.42	**85.14±0.20**
CIFAR-10	256	47.92±0.20	40.99±0.41	**53.78±0.36**	47.49±0.22	40.65±1.45	42.80±0.44	52.77±0.45
	1024	51.62±0.25	49.38±0.77	60.65±0.14	51.39±0.46	42.86±0.88	16.22±12.44	**63.99±0.47**

So, remember that we motivated this idea as a test of whether improved interpolation also improves representation learning performance by mapping similar datapoints close together in latent space. One way we can test this is by training a single-layer classifier, a logistic regressor, on the latent codes learned by each autoencoder on various common datasets. Note that we do not optimize the autoencoder's weights with respect to the classification objective - we treat the weights as fixed, produce the latent codes, and only optimize the classifier's parameters to improve classification performance given this fixed feature representation. We found that the simple autoencoder combined with ACAI giave the best results in nearly every case, across three datasets and two latent dimensionalities. This suggests that indeed there may be some link between representation learning performance and interpolation ability, that interpolation suggests some useful structure in the latent space.



References

Gulrajani, Metz, and Raffel, "*Training Implicit Generative Models by Unrolled Gradient Descent on Adversarial Divergences*"

Gulrajani, Raffel, and Metz, "*Towards GAN Benchmarks Which Require Generalization*"

Nagarajan, Raffel, and Goodfellow, "*Theoretical Insights into Memorization in GANs*"

Berthelot*, Raffel*, Roy and Goodfellow, "*Understanding and Improving Interpolation in Autoencoders via an Adversarial Regularizer*"

Thanks!

Here are some references for the papers I talked about today.