Neural Importance Sampling

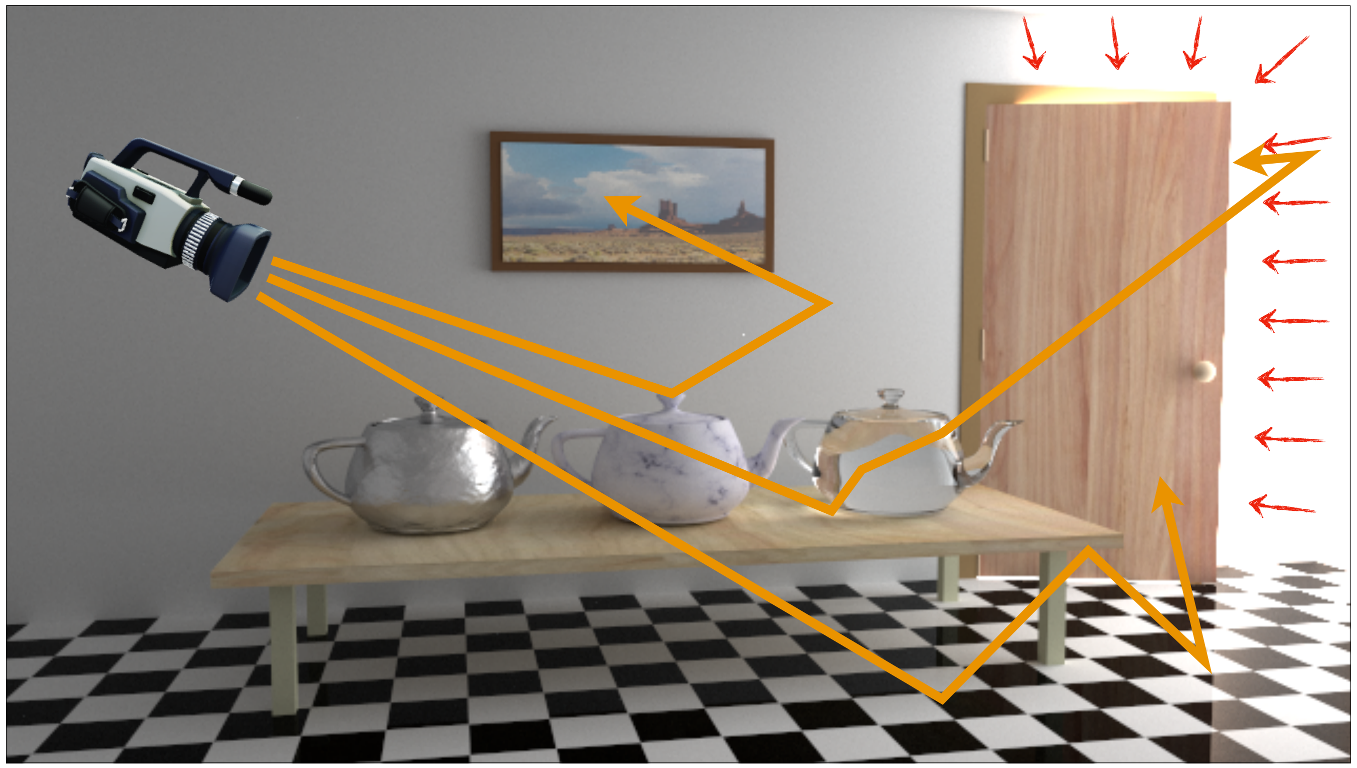
Thomas Müller
Brian McWilliams
Fabrice Rousselle
Markus Gross
Jan Novák

Affiliation: NVIDIA

Work done while at: ETH zürich Disney Research cgl computer graphics laboratory

Hello, I am Thomas, and I'll present our work on using neural networks for importance sampling in Monte Carlo integration.

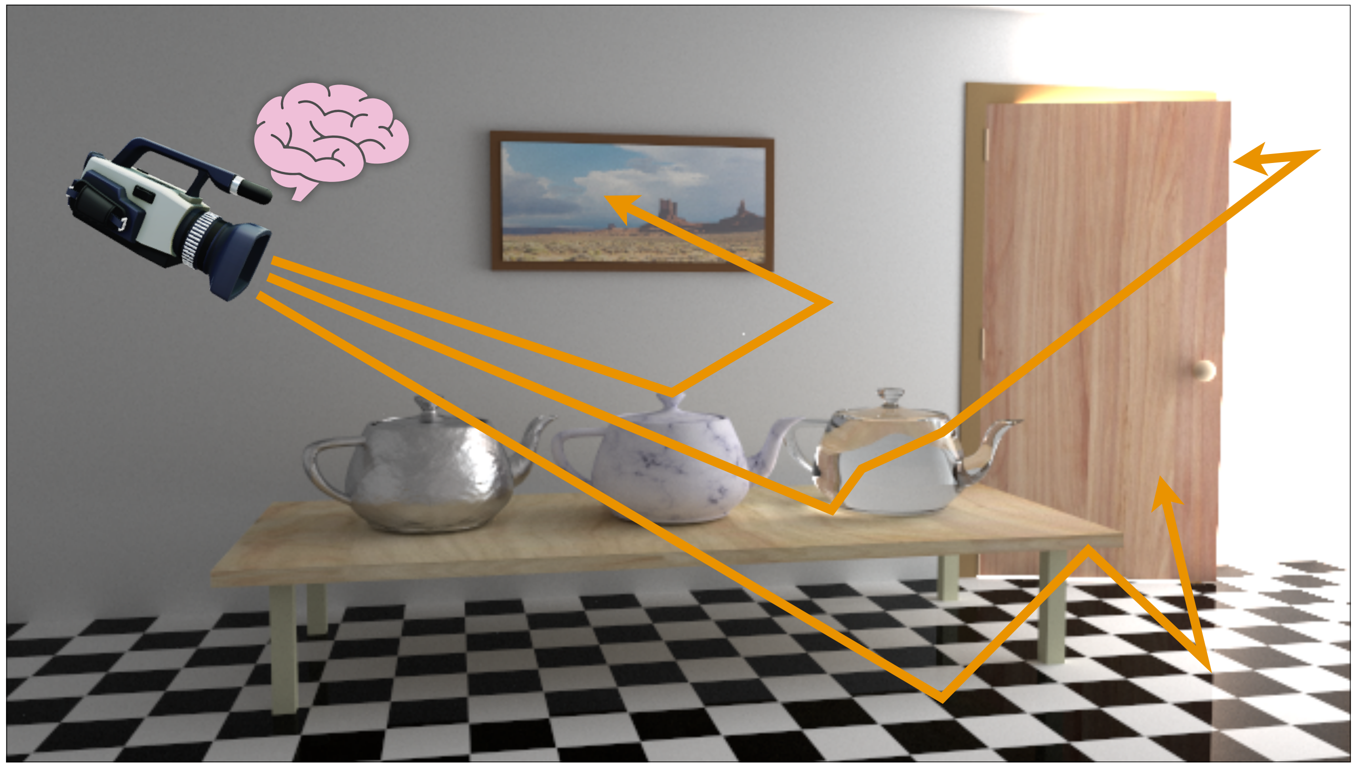
Let me start with a small example.



Suppose we want to render this image here with path tracing. All the light enters the room from this door opening, so we'd like *all* paths to go through it, but when you use standard path tracers that's typically not what happens... most paths just bounce around pretty much aimlessly and never make it through the door, and this leads to a lot of noise.

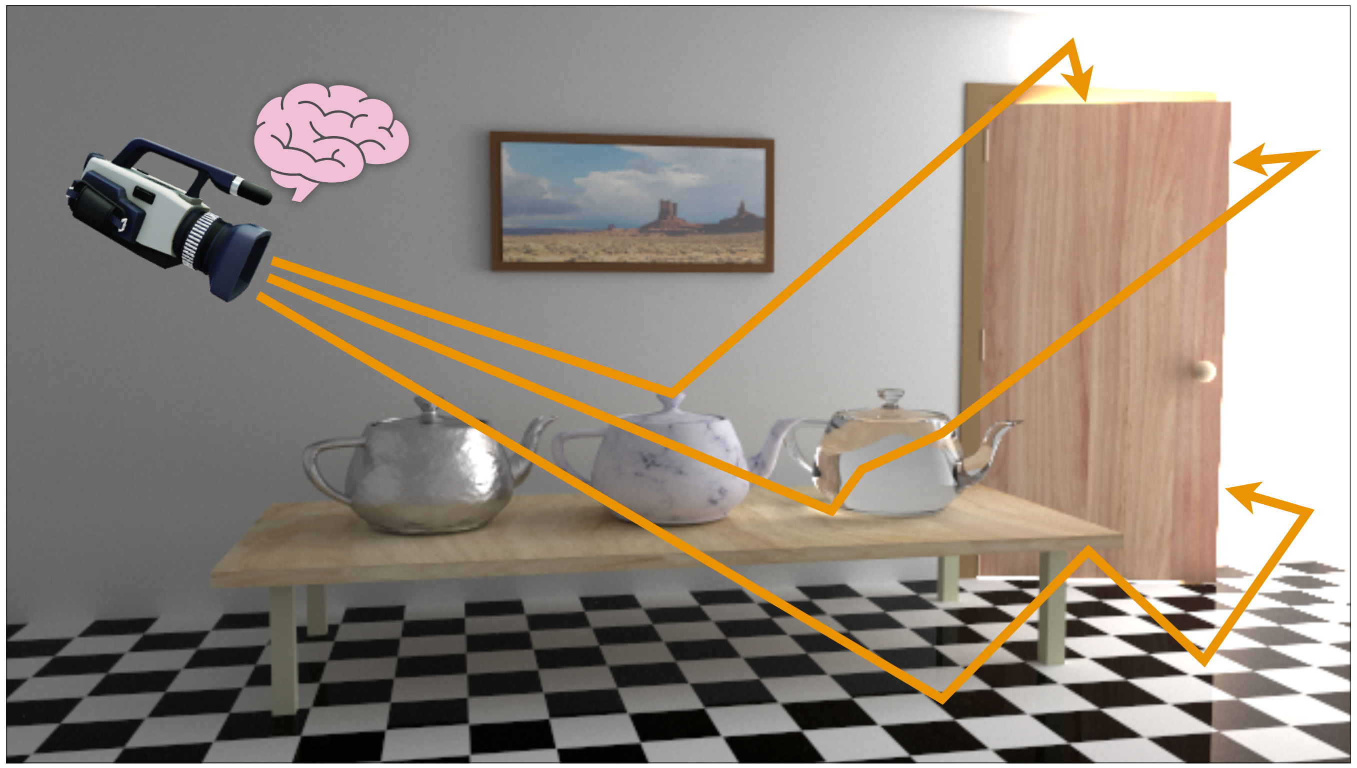
Render time: sometimes >100 cpu-hours

It's actually not uncommon that it takes hundreds of cpu hours for difficult scenes like this to converge to any sort of reasonable noise level, and that's clearly not okay.

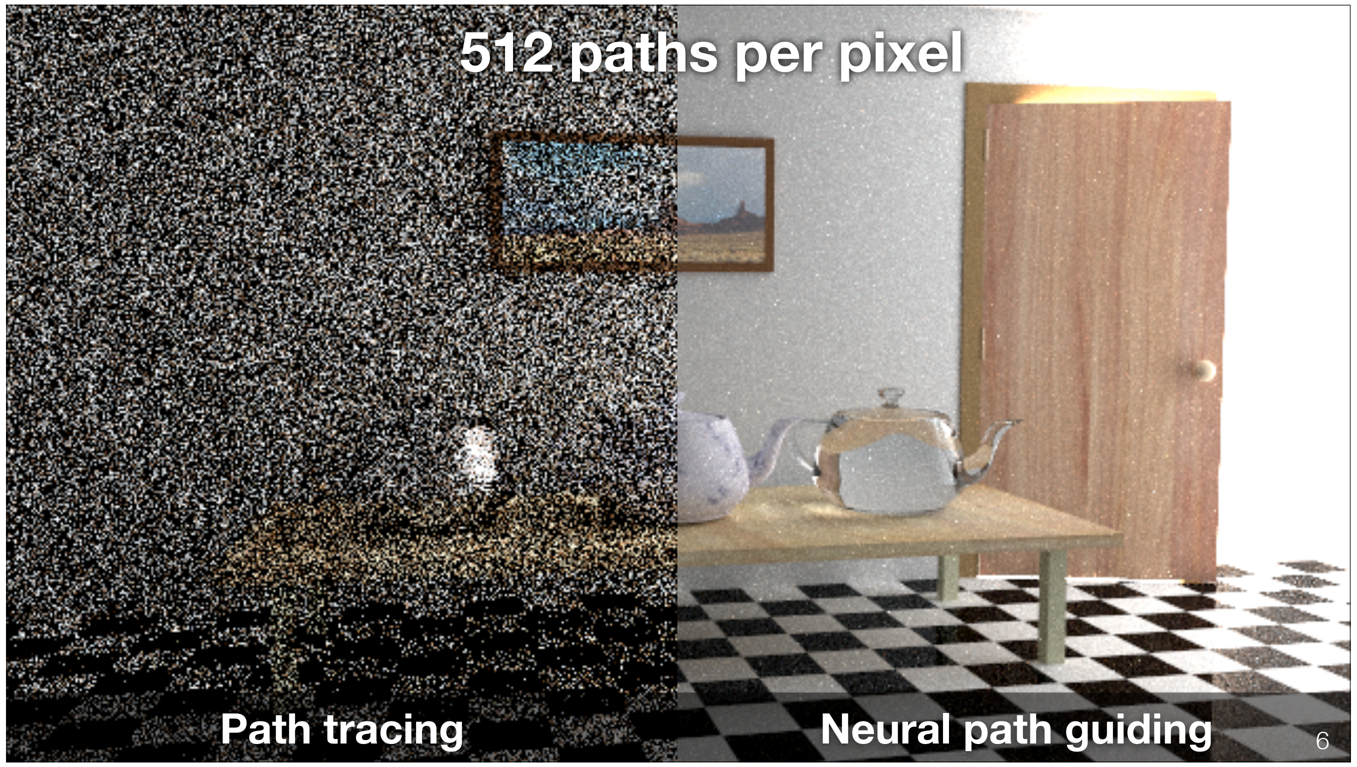


So in this talk, we'll look at one particular way to reduce the noise.

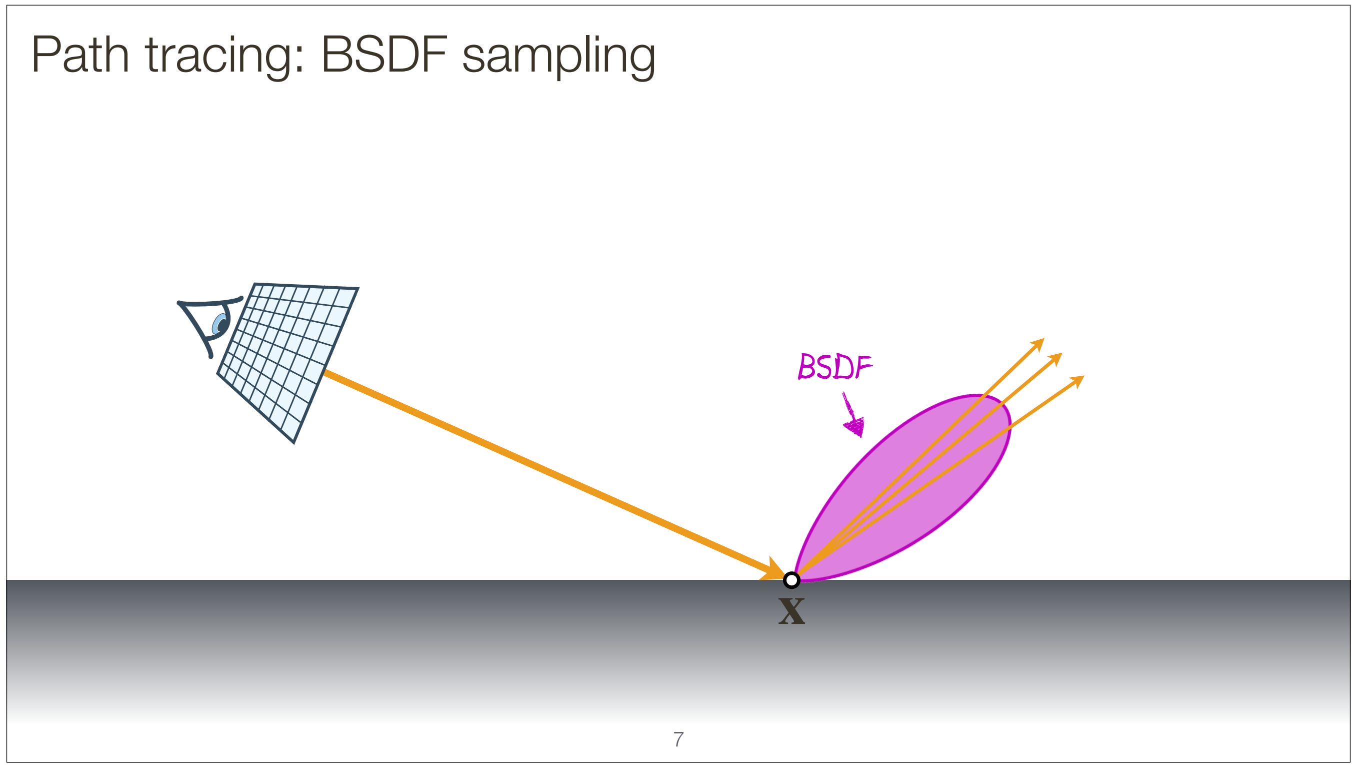
We'll look at how we can train a *neural network* from these not-so-optimally traced paths in such a way...



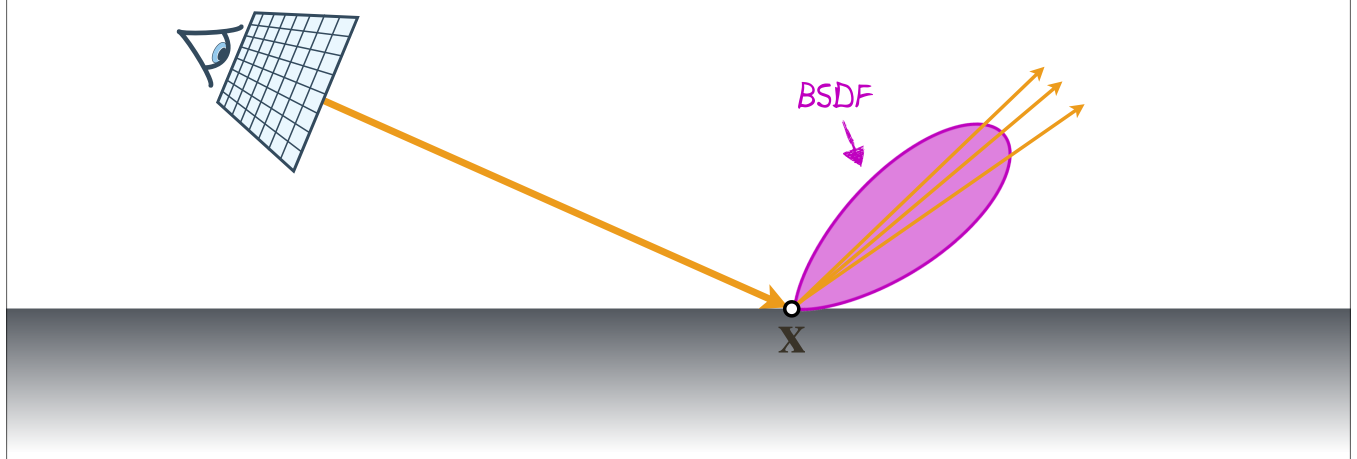
...that the network learns to *guide* future paths to the right places. Just to motivate why we bother doing this...



...this is the kind of difference in noise you can expect between regular path tracing and our neural path guiding.



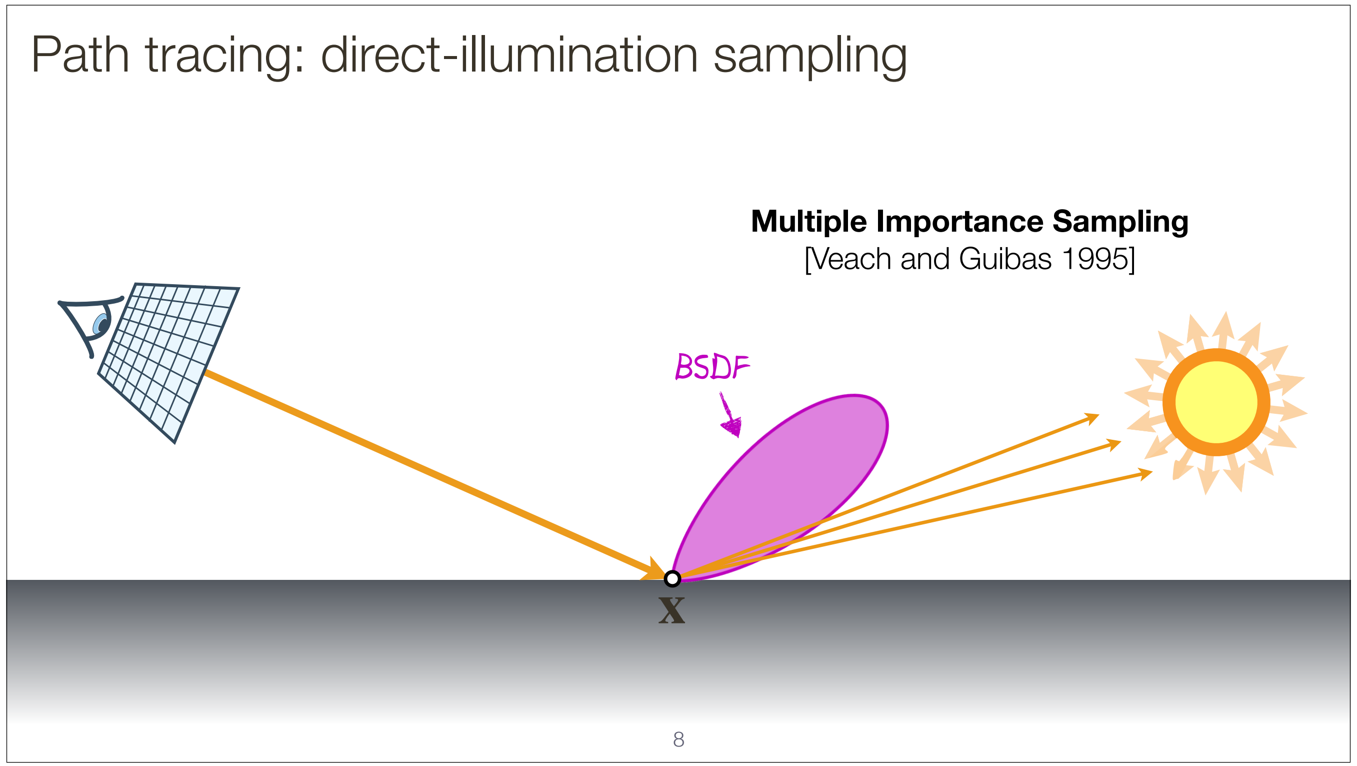
Path tracing: BSDF sampling



Let's have a more detailed look.

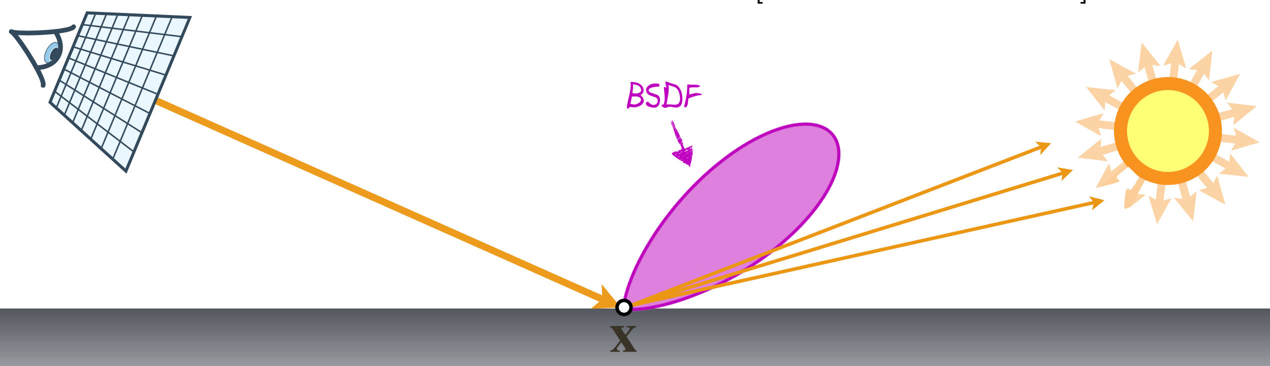
Whenever a path hits a surface, we need to sample the direction to continue the path in.

Standard path tracers sample either from the BSDF...



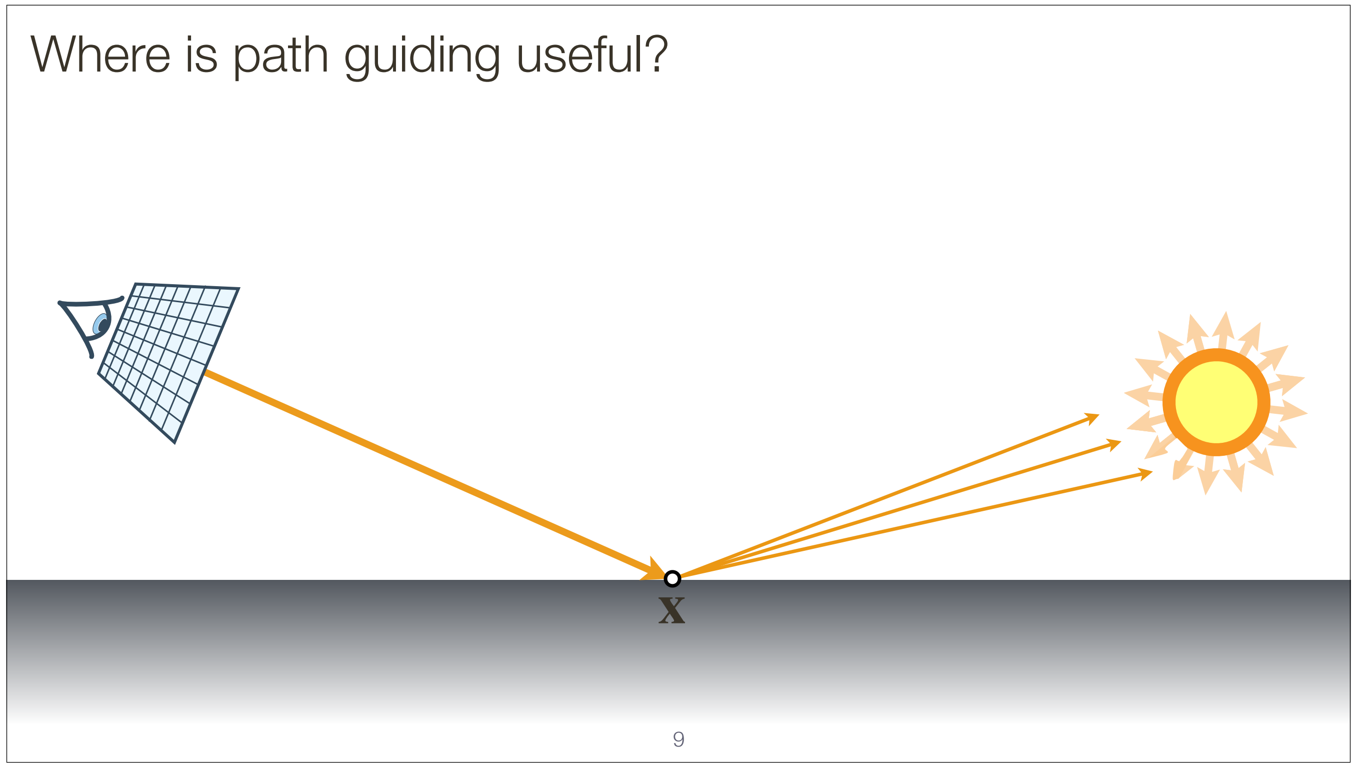
Path tracing: direct-illumination sampling

Multiple Importance Sampling
[Veach and Guibas 1995]

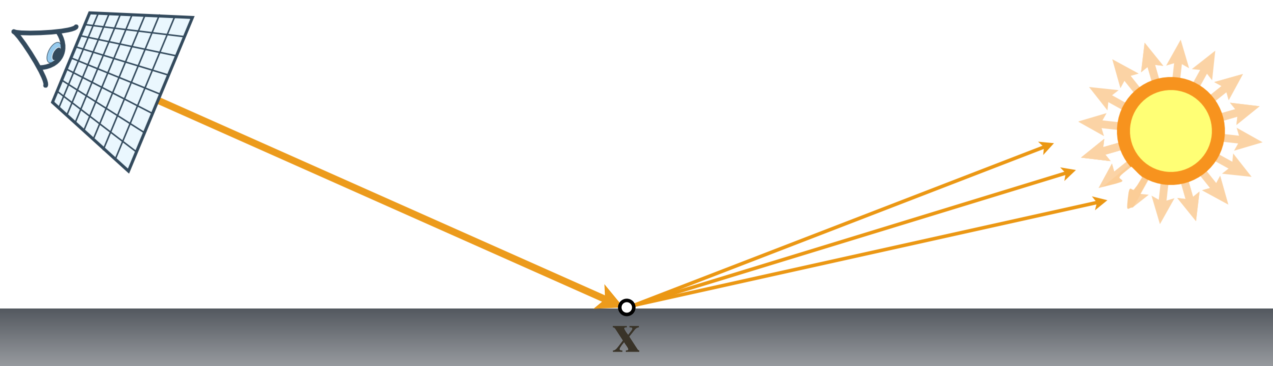


...or, they connect directly with a randomly selected light source. This is called next event estimation.

BSDF sampling and next event estimation are then typically combined with multiple importance sampling, and that's pretty much the standard path tracer that you see in most places.

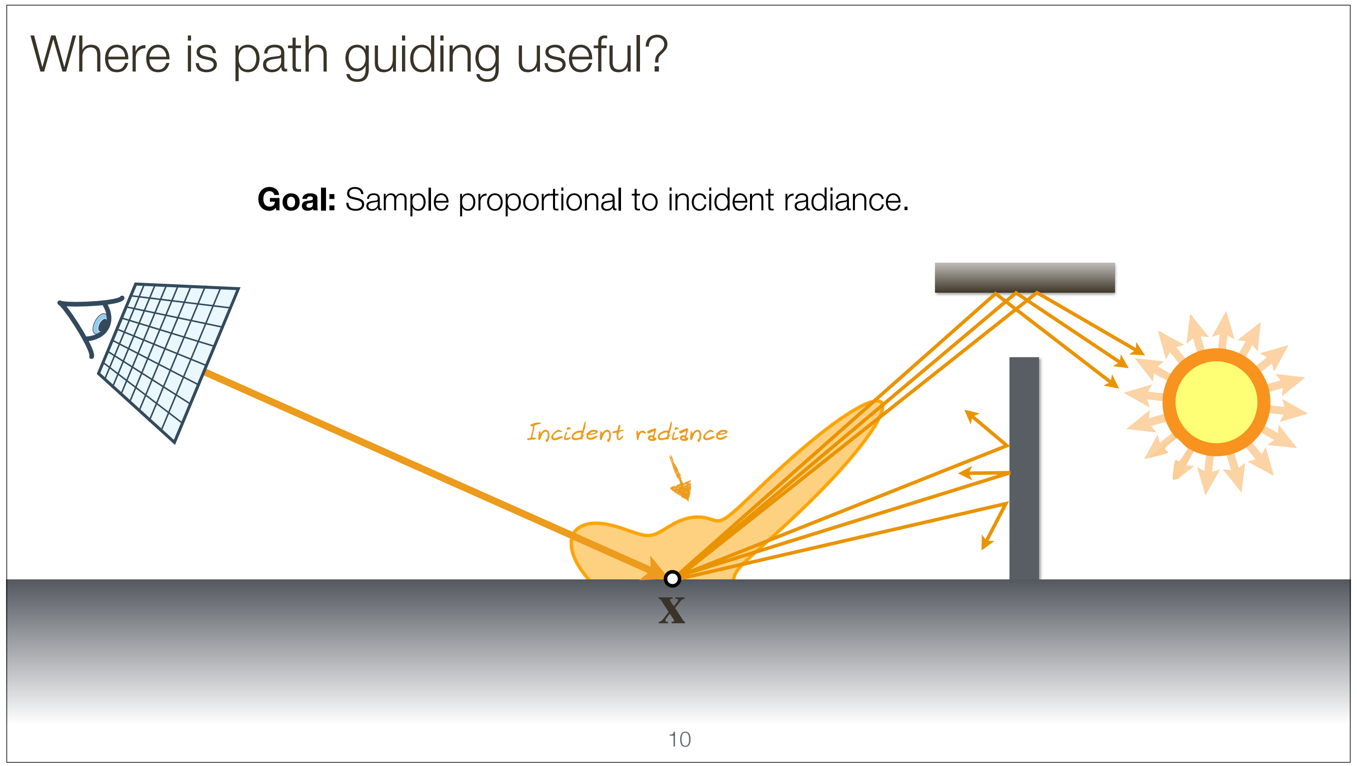


Where is path guiding useful?



Now let's look at where this path tracer breaks.

Suppose we place an occluder right in front of the light source...



Where is path guiding useful?

Goal: Sample proportional to incident radiance.

...and we put a reflector here.

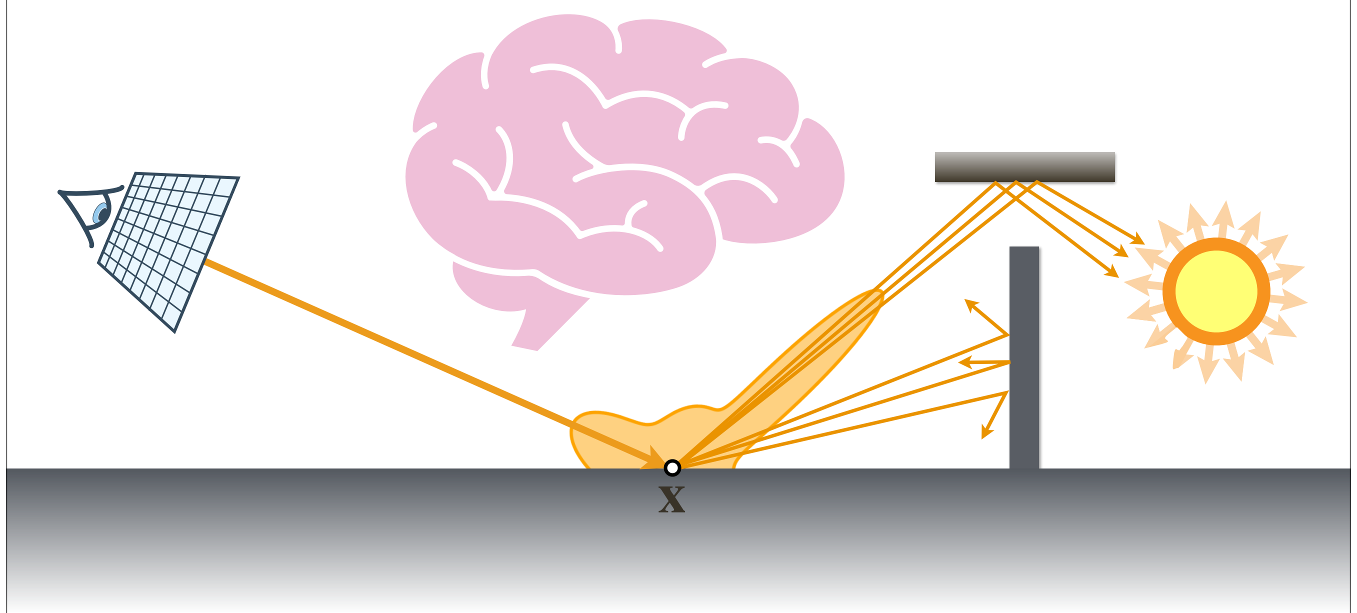
First of all, the occluder blocks the direct paths to the light source, and secondly, by adding the reflector, suddenly new light paths contribute illumination *indirectly* to our shading location at x.

So if we plotted the *incident radiance distribution* at x, it might look something like this.

Our goal is now to somehow learn this distribution *on-line during rendering* from the paths that we trace, and to then use this learned distribution to *guide*---in other words: importance sample---future paths.

Previous work on this sort of thing---in general, the whole family of path-guiding techniques---used all kinds of hand-crafted data structures and heuristics. In contrast to that, *our* goal is to use a neural network.

Where is path guiding useful?

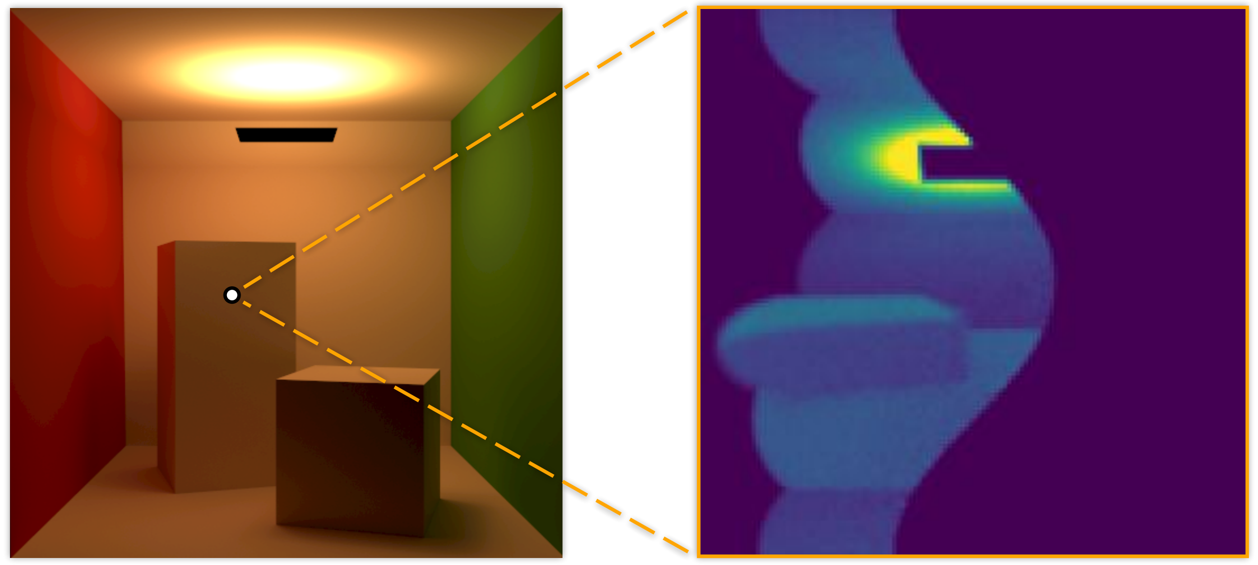


Why would we do this?

Neural networks are great at learning *really high*-dimensional functions---like natural images with millions of pixels---but we're just trying to learn the incident radiance, which is only 5-dimensional, so for neural network standards that's pretty low.

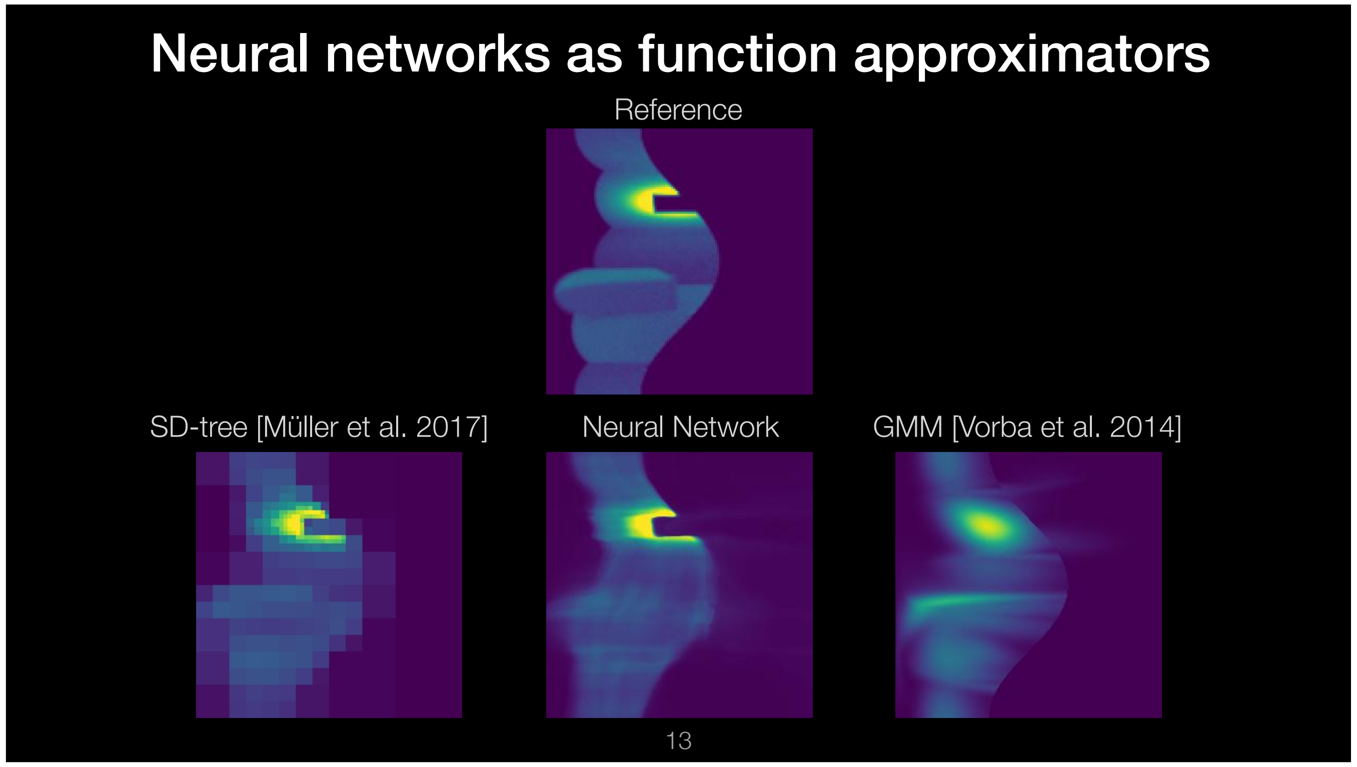
So for this particular task... is it even worth it? Are neural networks better than traditional approaches? To answer this question, let's compare the networks against existing algorithms.

Learning incident radiance in a Cornell box

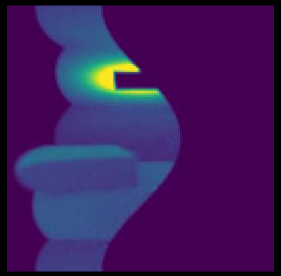


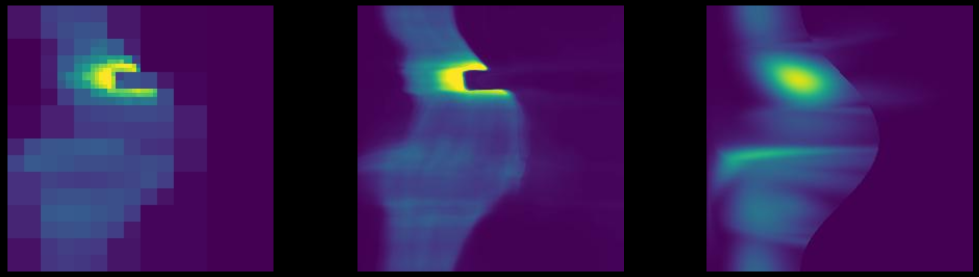
Let's look at what happens when we learn the *incident radiance* in a simple Cornell box scene, only that in this scene we *flipped* the light upside-down such that it shines at the ceiling rather than towards the bottom, just to make the problem a little more difficult.

So at each *position* within the Cornell box---like here indicated as the white dot---there is a corresponding 2-dimensional directional distribution of incident radiance. The goal is to learn this function as accurately as possible from a bunch of noisy Monte Carlo samples---in other words: from a bunch of randomly traced paths.



Neural networks as function approximators


Reference



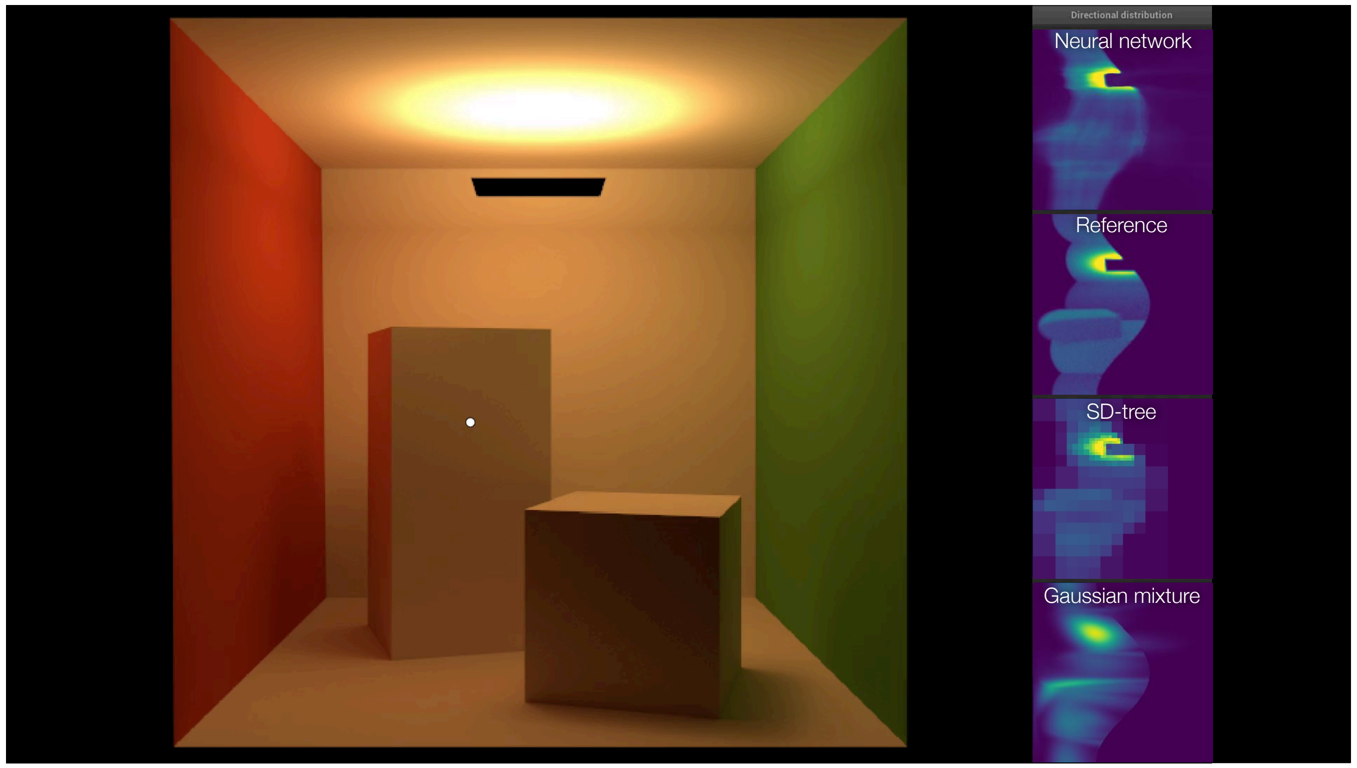
Let's first look at what the the SD-tree from Müller et al. learns, which is one particular existing path-guiding technique. It's learned approximation looks something like this. It does retain the general shape of the distribution, but it is also overall blurry and low-resolution.

In contrast, *this* is what the gaussian mixture model of Vorba et al.---another path-guiding technique---learns from the same samples. It is a lot smoother overall, but it is even blurrier than the SD-tree.

Now look at what a deep neural network learns from the *same* noisy samples as the other approaches. Let me emphasize that it's not trained from a large data-set, but only from the *same* small number of light paths.

It's actually a lot sharper than the other approaches, and it follows the reference quite a bit more faithfully, which is a really promising result. It shows that neural networks can outperform these existing data-structures in terms of accuracy for a relatively small training data set!

Now let's look at the spatial variation.



[See supplementary video.]

On the right, you see the same distributions as before and on the left you can see the Cornell box along with the position for which we visualize the radiance distribution.

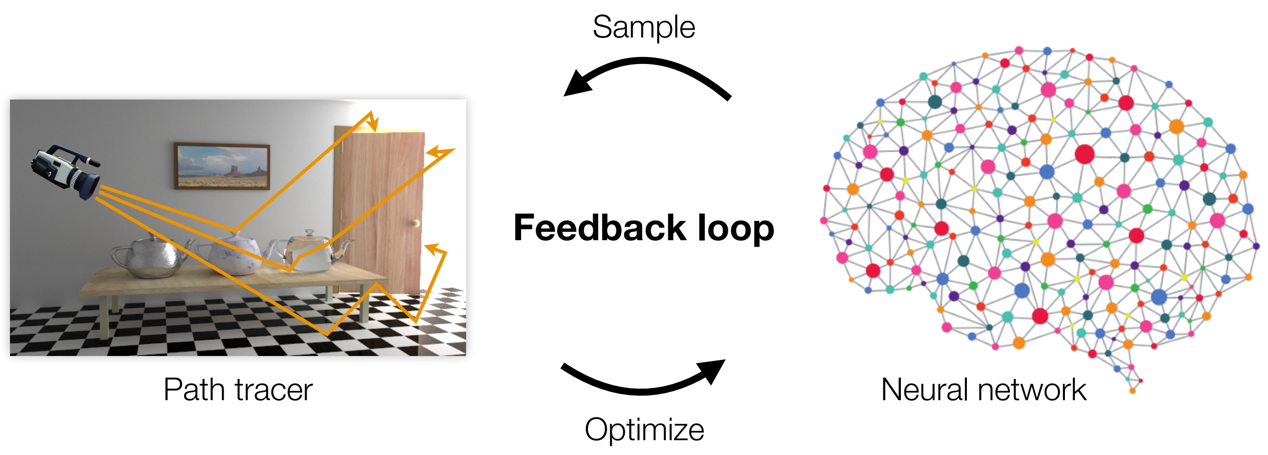
I will now start to animate the position within the Cornell Box---the white dot. Please have a close look at the distribution the neural network learned---it's in the top right---and compare how smoothly it varies against the other approaches.

...

So, in general, the neural network not only approximates the reference most accurately, but in contrast to the other approaches it also learns a continuously varying function. This is because all the previous approaches subdivide space either as a tree or some discrete data structure, whereas the networks simply take the spatial coordinate as input and learn by themselves how to map this position to the right directional distribution, rather than relying on ad-hoc subdivision heuristics.

In this sense, I would say the networks are actually *more* principled than the other approaches.

Neural path guiding overview



Okay, so we established that neural networks show promise---they seem to be able to learn really good representations---but how do they fit into the rendering process?

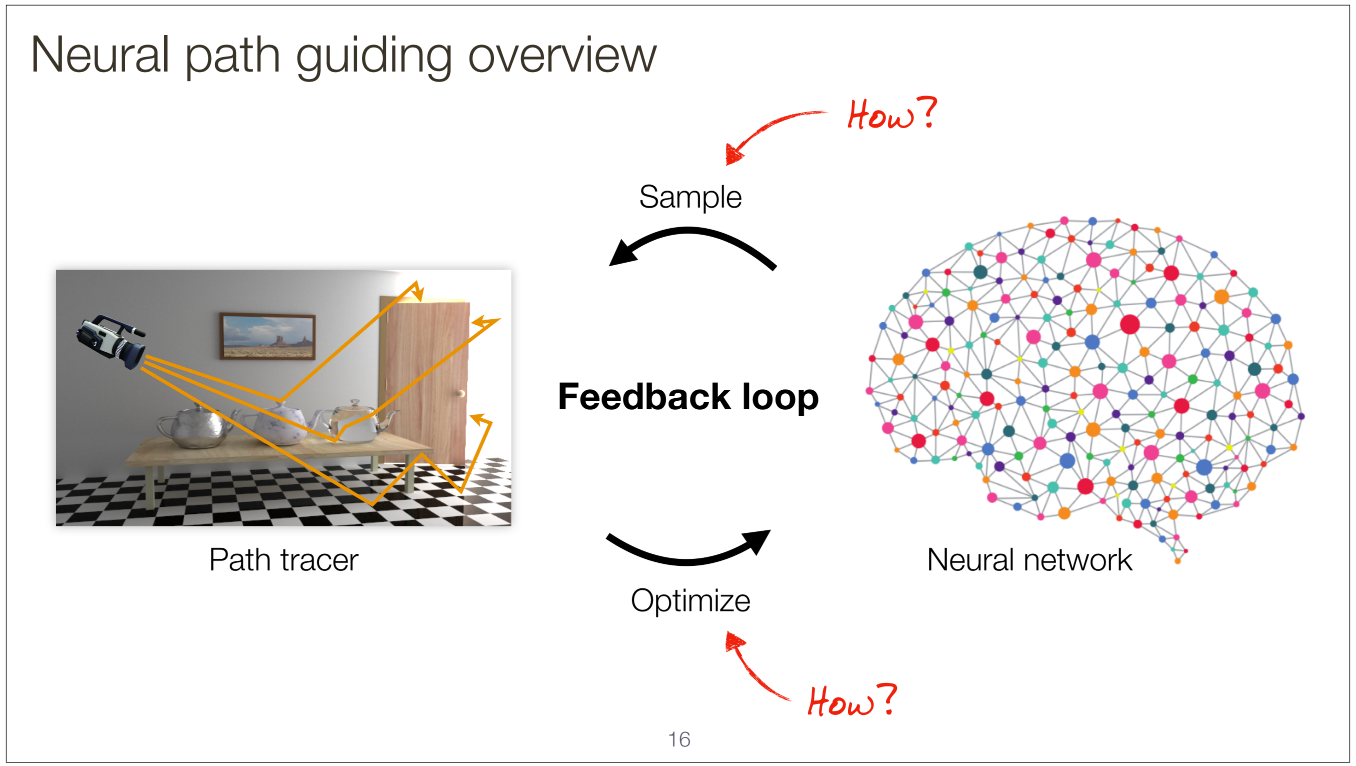
Well, the way we do it, is we begin by initializing our neural network randomly and then generating a bunch of initial light paths guided by this freshly initialized neural network.

Even though these paths may be very noisy, we can still use them to optimize the neural network.

We then use the (hopefully better) neural network to generate more guided light paths, and we use those for further optimizing the neural network.

This creates a feedback loop where the better paths have less noise, resulting in both a nicer image, *but also in better training data*.

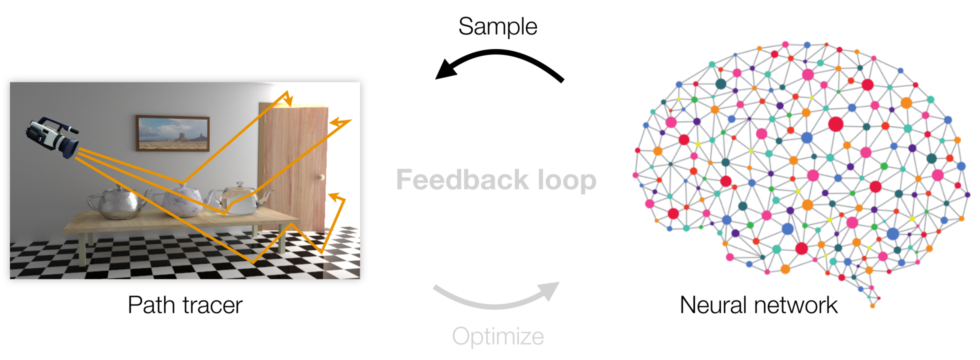
So this all sounds good in theory, but we have this problem of...



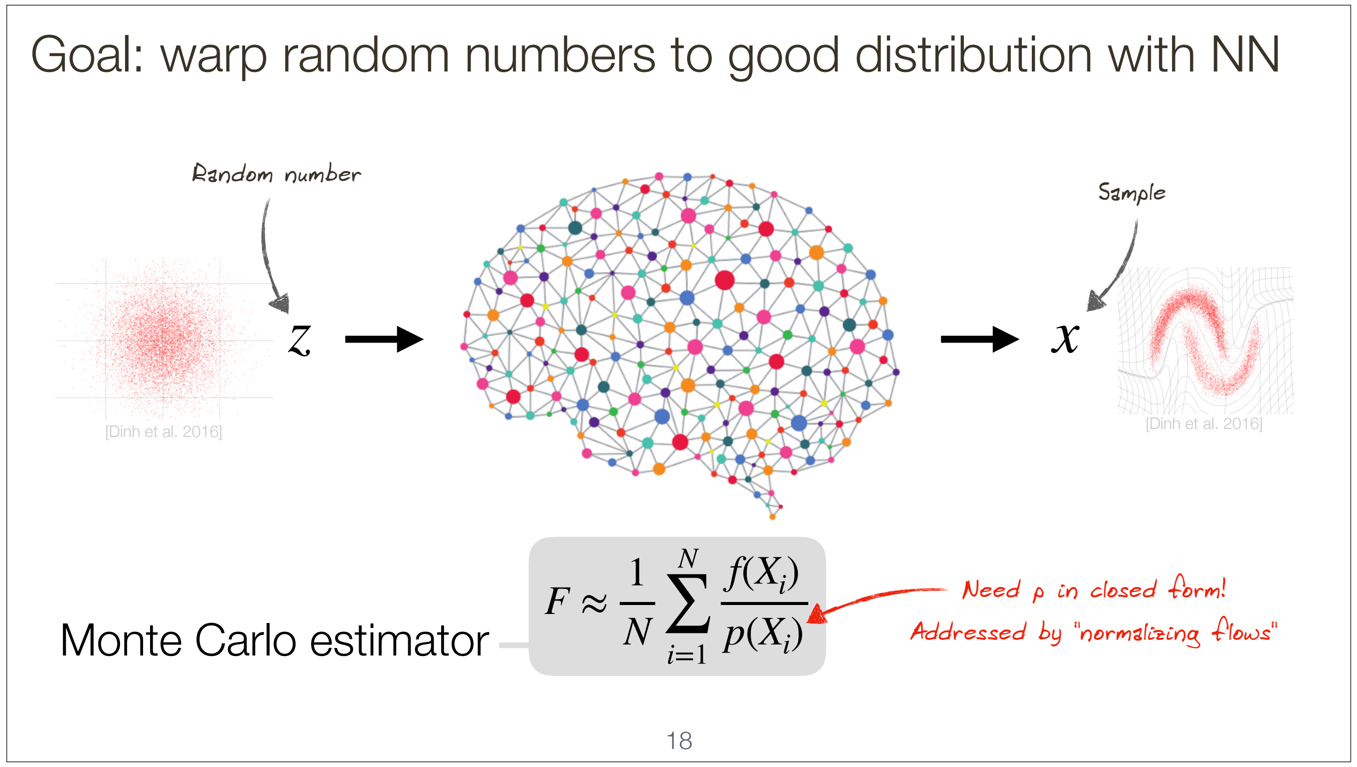
Neural path guiding overview

...not really knowing how to do either sampling *or* optimization. Machine-learning literature has very little to offer on this questions, because this particular application of neural nets is relatively new. Answering the question of how to do sampling and optimization within a Monte Carlo context---that's what our paper really is about and what I'll talk about next.

How to draw samples?



I'll start with the question of how to draw samples.



Goal: warp random numbers to good distribution with NN

Monte Carlo estimator

$$F \approx \frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{p(X_i)}$$

Need p in closed form!
Addressed by "normalizing flows"

18

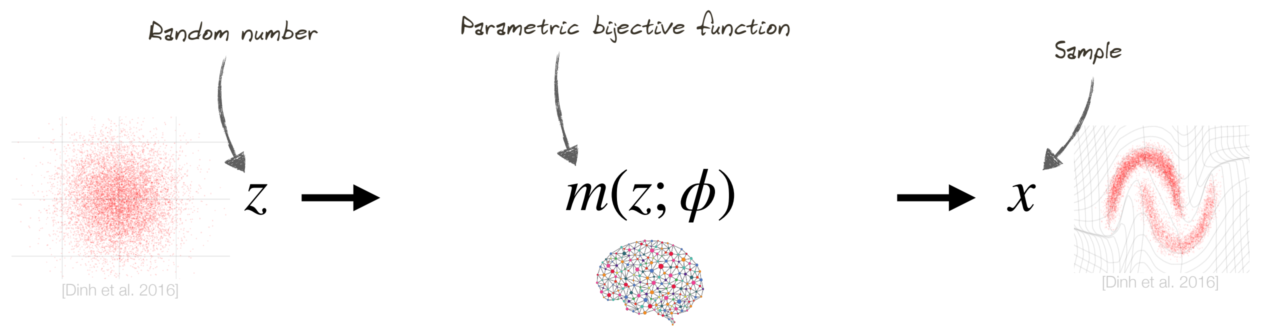
Traditional generative models look like this: we have a latent random variables **z** as input to our neural network, which transforms it into samples **x**. We can postulate the distribution of **z** however we like---for example Gaussian---and we need to somehow optimize the network such that the distribution of **x**---which is the distribution of **z** after having been transformed by the network---matches some target distribution that we desire.

This approach works fine in many cases, but to be able to use it for importance sampling within Monte Carlo, there is a challenge that we need to overcome. Here's the formula of a Monte Carlo estimator: in order to use it, we need to not only be able to *draw* samples, but we also need to *evaluate* the probability density of samples!

But the kind of architecture that you see above does *not* allow evaluating the probability density. The distribution of **z** may be known, but after being piped through an *arbitrary* neural network, the distribution of **x** is generally difficult to obtain, so we can not use it for Monte Carlo!

Thankfully, there has been some work in the machine-learning community on architectures that *allow* evaluating **p(x)**: those based on so-called "normalizing flows", and the key idea behind normalizing flows is the following.

Parameterizing a bijection allows using the change-of-variable formula


Random number
Parametric bijective function
Sample
$z \rightarrow m(z;\phi) \rightarrow x$
[Dinh et al. 2016]
[Dinh et al. 2016]

$$p(x) = p(z) \cdot \left| \det \left(\frac{\partial m(z)}{\partial z^T} \right) \right|^{-1}$$

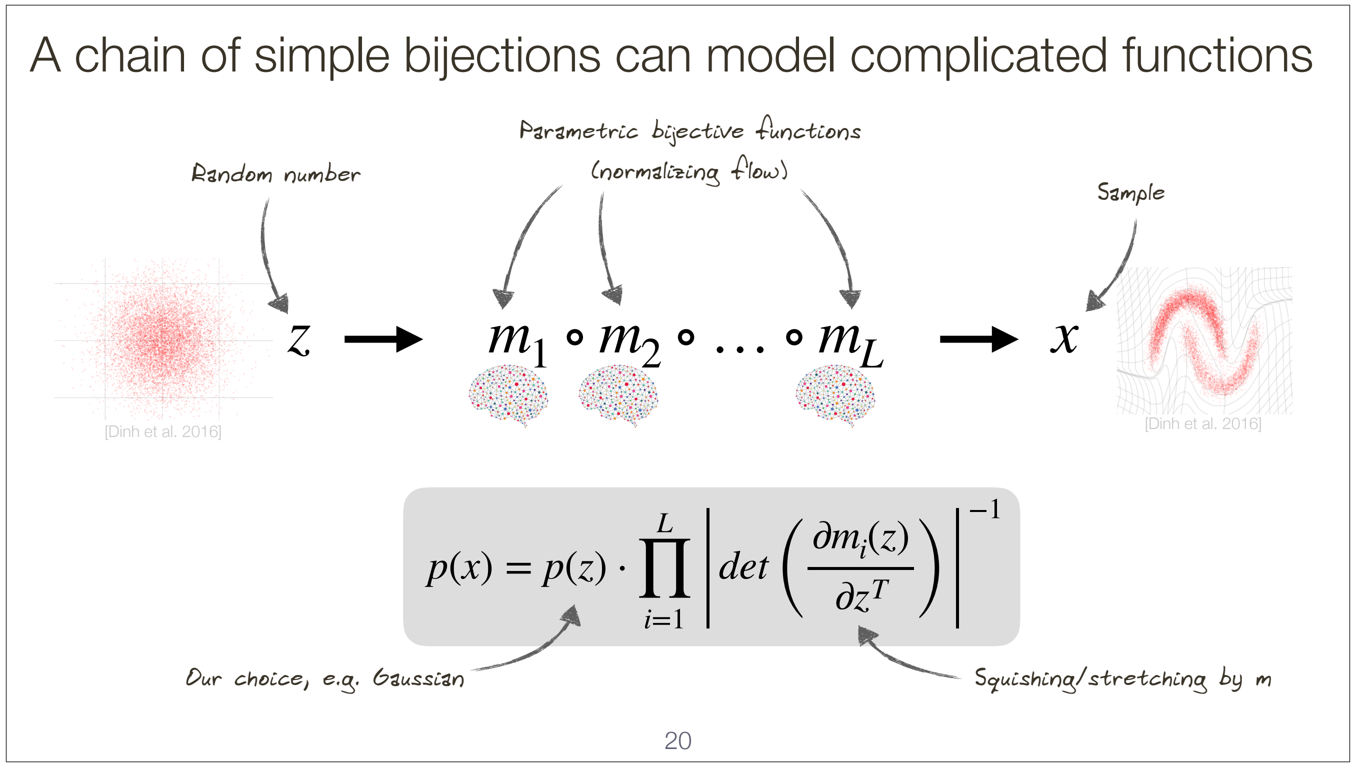
Our choice, e.g. Gaussian
Squishing/stretching by m

Instead of piping **z** directly through a neural network, instead, it is piped through a *parametric* bijective function---let's call it m----the *parameters* of this function **phi** are the thing that's controlled by the neural network.

And because m is bijective, if it's differentiable, we can then express **p(x)** in terms of **p(z)** and the **m** via the change-of-variables formula. **p(x)** is **p(z)**, multiplied by the *inverse Jacobian determinant of* **m**.

We can pick **p(z)** however we like, and the Jacobian determinant captures by how much **m** *squishes and stretches* space locally, which is why it affects the density.

So how do we use this scheme it in practice? Well, to make the scheme practical, we need to choose a bijection **m** that has two key properties: *first* it needs to be expressive such that it can capture complicated light fields and *second* it needs to have an efficiently computable Jacobian determinant such that we can use this formula here. Getting both of these properties at the same time is surprisingly difficult, so a good strategy is to *compose multiple less-expressive but easy to handle functions* where the composition recovers good expressivity.



A chain of simple bijections can model complicated functions

Random number

Parametric bijective functions (normalizing flow)

Sample

$$z \longrightarrow m_1 \circ m_2 \circ \ldots \circ m_L \longrightarrow x$$

[Dinh et al. 2016]

[Dinh et al. 2016]

$$p(x) = p(z) \cdot \prod_{i=1}^{L} \left| det \left(\frac{\partial m_i(z)}{\partial z^T} \right) \right|^{-1}$$

Our choice, e.g. Gaussian

Squishing/stretching by m

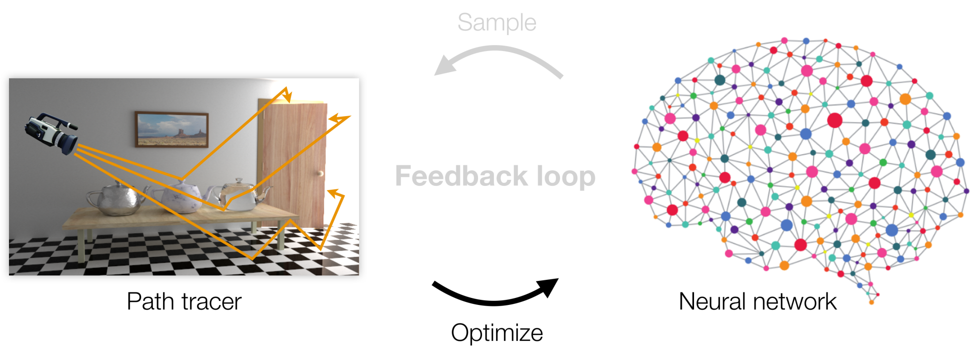
Here *each of the parametric bijective functions* gets its own neural network parameterizing it. The change-of-variables formula only changes slightly: we simply multiply up all the squishing/stretching terms from the individual bijections.

With this tool in hand, we now have to choose what kind of bijective function we use.

We actually spent quite of time in the paper to introduce a novel function that really works well here---and the machine-learning community has already picked it up extended it further---but this is kind of a detail and not so much related with computer graphics, so I rather want to focus on the bigger picture for the rest of this talk. So the bottom line here is---this normalizing-flow architecture allows us to draw samples for unbiased Monte Carlo integration, *driven by a neural network,* and now...

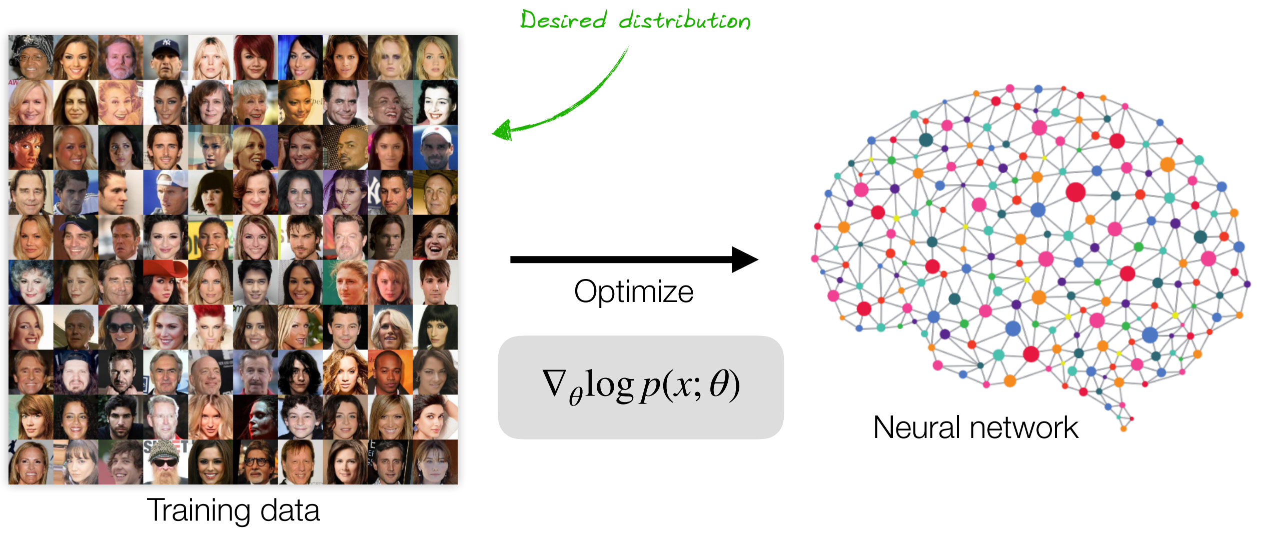
[The following "bonus" slides describe our choice of m]

How to optimize?



...let's have a look at how we can then *optimize* our networks *from* the drawn Monte Carlo samples.

Training with data from the correct distribution is simple



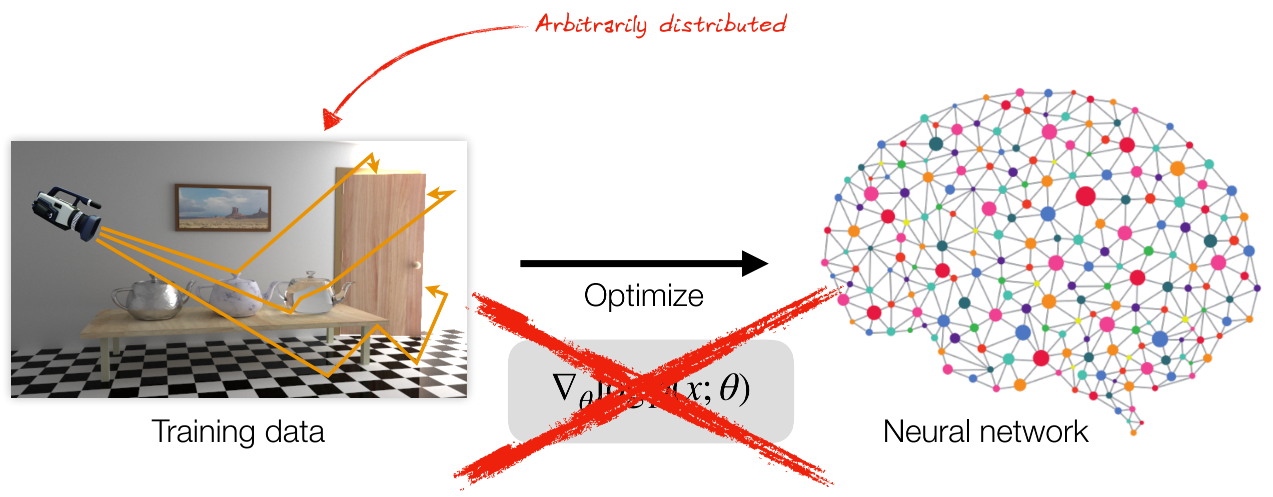
$$\nabla_\theta \log p(x; \theta)$$

Let's first look at how neural networks are *typically* trained in the wild.

Usually, training data consists of examples that are drawn from the distribution that we want to learn and the network's goal is to *imitate* that distribution. For example, if the goal is to generate realistic faces of celebrities, then the training data is... a set of faces of celebrities.

And then one would, for example, optimize the network to maximize the log-likelihood of the data. So in the framework of neural networks, we would do gradient descent on the log-likelihood. Since our sampling framework that we just looked at *directly* gives us the probability density of **x** we can compute this gradient of this log-likelihood analytically using backpropagation and directly use it to optimize the network.

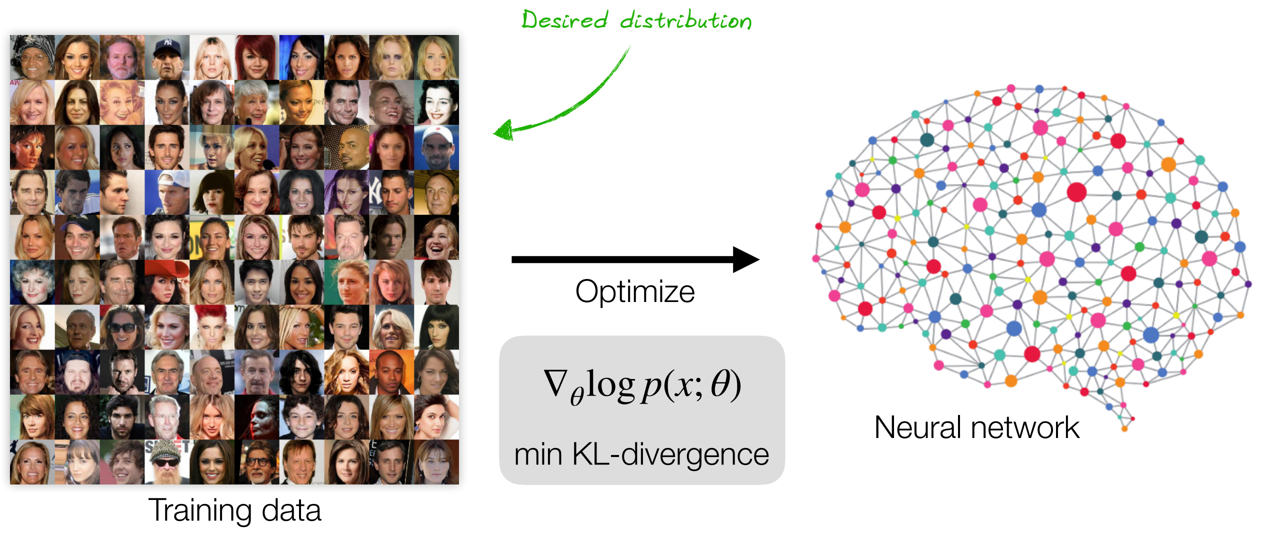
Training from Monte Carlo samples requires careful weighting



But what happens when the training data is *arbitrarily* distributed such as the paths in a path tracer? These paths are generated with---for example---BSDF sampling so they don't actually follow the distribution we like to learn. Our goal is to take these samples that are *wrongly* distributed, along with their *Monte Carlo weights*, and to use *those* to train our neural network.

If we were to naïvely maximize the log-likelihood as before, then our networks would just learn to replicate the *sub-optimal existing* distribution of paths that we *can* already sample from, so this doesn't bring us any further.

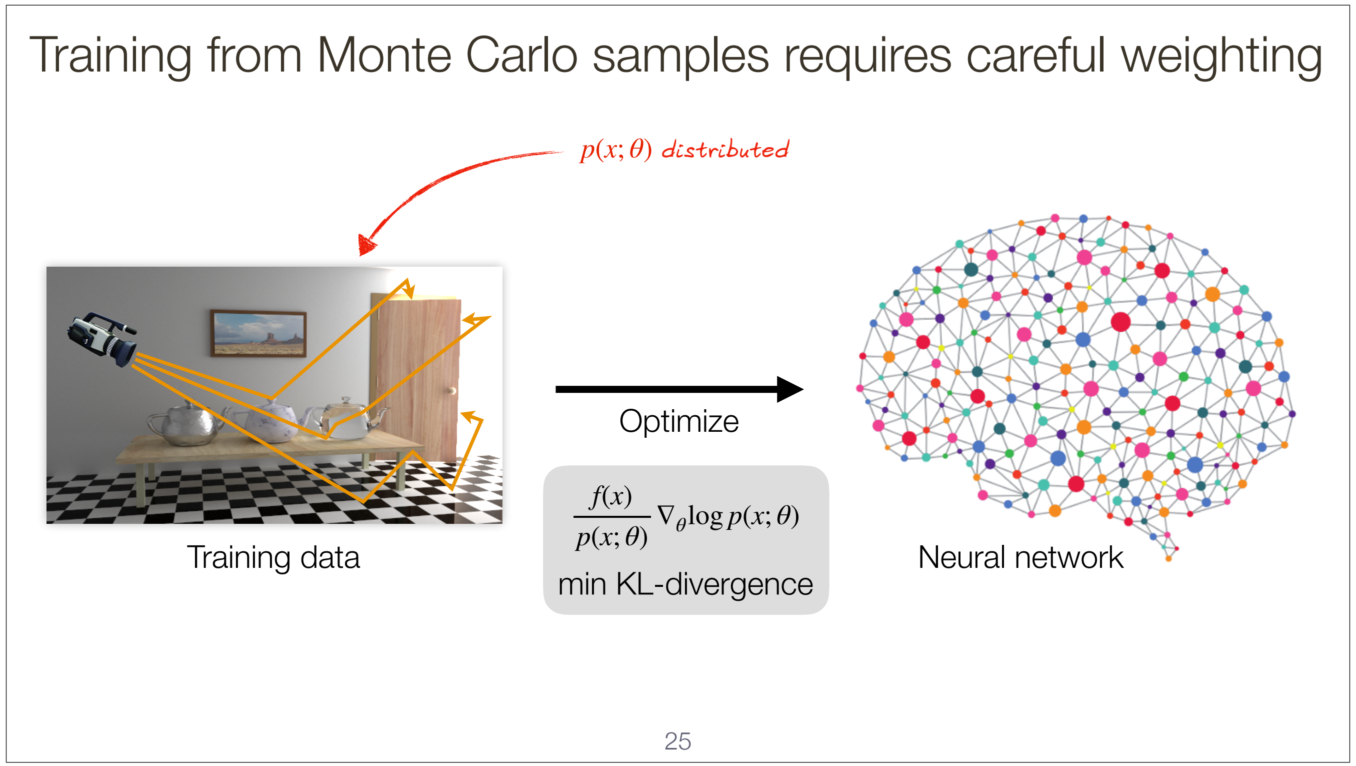
Training with data from the correct distribution is simple



To go further from here, let's look at the problem from a slightly different perspective.

When we maximized the log-likelihood for the case where we have data that's distributed as desired, what we did was actually *equivalent* to *minimizing the Kullback-Leibler-divergence* between the desired distribution and whatever the neural network has learned.

Our goal is now to *also* minimize the Kullback-Leibler-divergence, but for the case where the training data is *not* distributed correctly.



Training from Monte Carlo samples requires careful weighting

$p(x;\theta)$ distributed

Training data

Optimize

$$\frac{f(x)}{p(x;\theta)}\nabla_\theta \log p(x;\theta)$$

min KL-divergence

Neural network

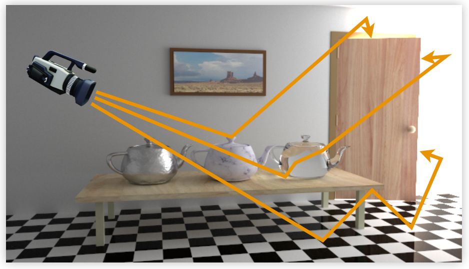
This, in turn, turns out to be possible. I'll spare you the derivations---they're in the paper. What we get in the end is that when we have training data distributed according to an *arbitrary*, but *known* distribution, we *still* do gradient descent on the log-likelihood! Only, that the log-likelihood gradient must be weighted by the corresponding Monte Carlo estimates of the samples---the f-over-p term at the front of this formula---which in our case of path-tracing is, for example, the amount of radiance carried by each path.

So now we have a principled recipe for optimizing our neural networks from noisy paths...

But... at this point we might wonder... this KL-divergence---we ended up with it because we started with log-likelihood maximization. But is it really the thing we want to minimize in Monte Carlo integration? Is there maybe some other function we might want to minimize and that we can we make similar derivations for?

Training from Monte Carlo samples requires careful weighting

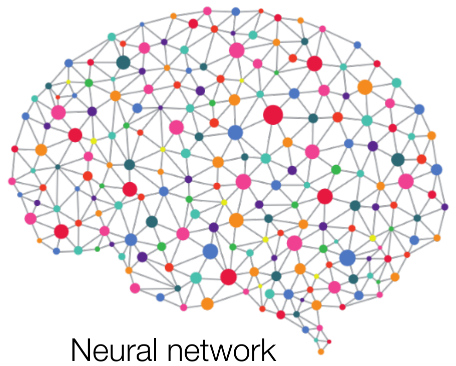
$p(x;\theta)$ distributed


Training data

$\xrightarrow{\text{Optimize}}$

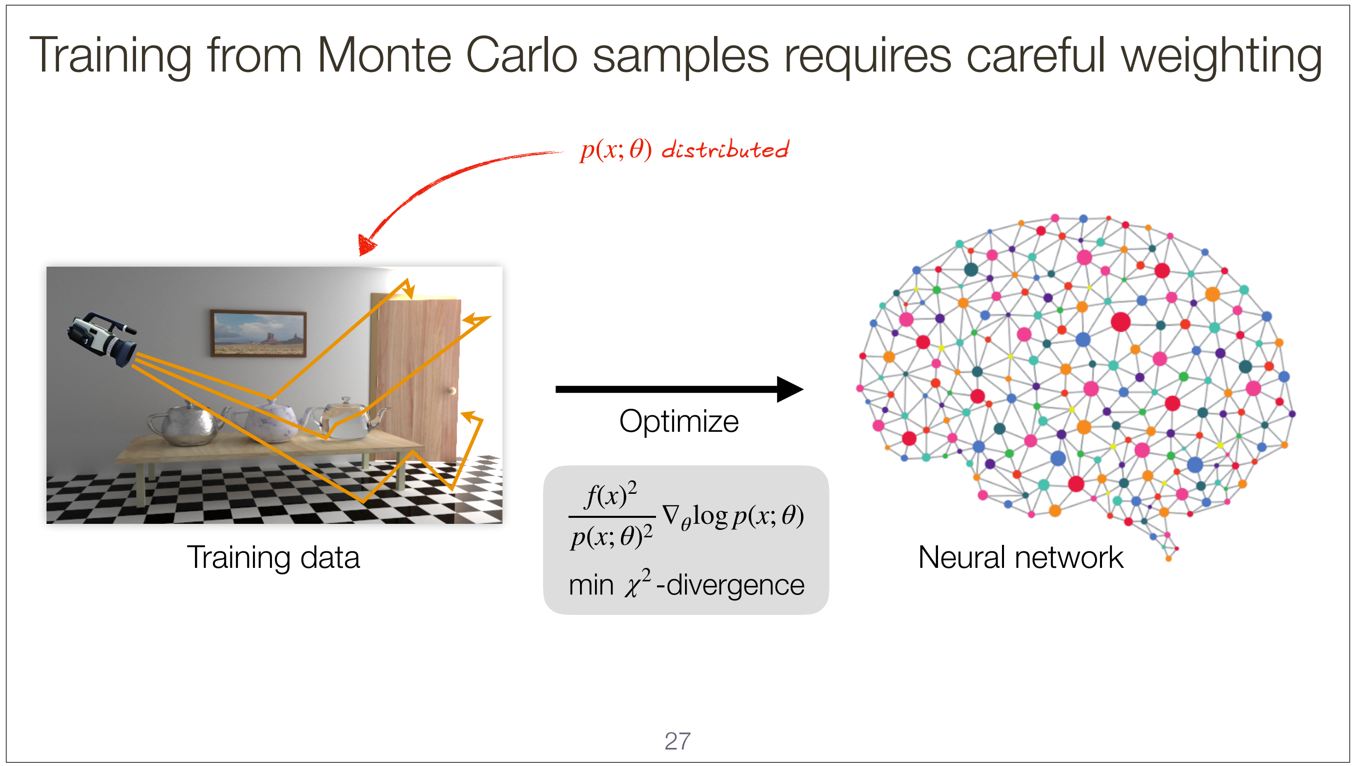
$$\frac{f(x)^2}{p(x;\theta)^2}\nabla_\theta \log p(x;\theta)$$

min variance


Neural network

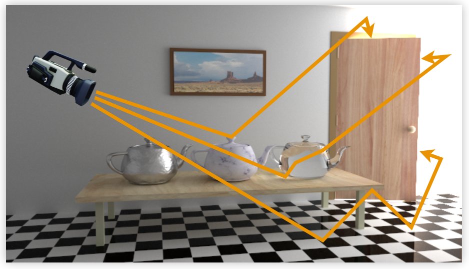
Maybe the most obvious choice would be to *directly* minimize the Monte Carlo *variance*... and it turns out, that this is also possible. Again, sparing you the derivation, this formula here pops out.

Interestingly, this formula *still* contains the log-likelihood, but this time weighted by the *squared* Monte Carlo estimate, so it uncovers a close relation between the variance and the KL-divergence---the only difference is the square around the Monte Carlo weight.



Training from Monte Carlo samples requires careful weighting

$p(x;\theta)$ *distributed*

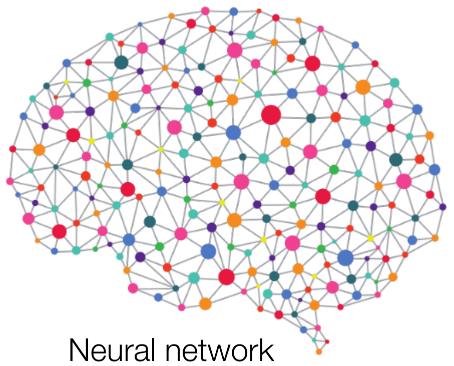


Training data

Optimize

$$\frac{f(x)^2}{p(x;\theta)^2}\nabla_\theta \log p(x;\theta)$$

min χ^2-divergence



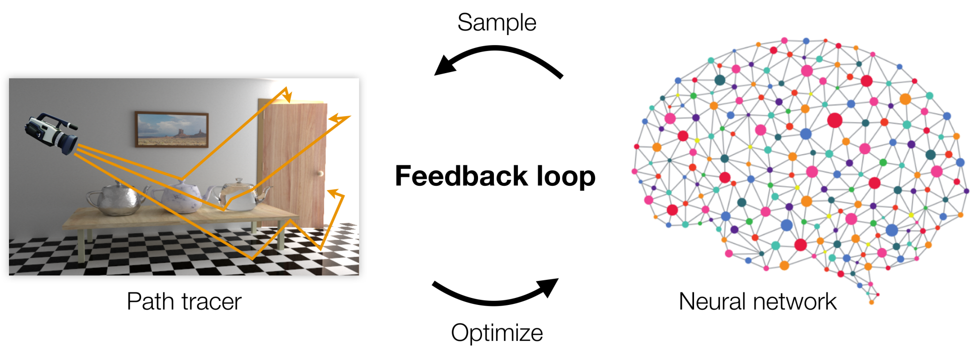
Neural network

Digging a bit deeper, minimizing the variance turns out to be equivalent to minimizing the chi-squared divergence, which is actually related to the KL-divergence, so there seems to be this whole family of loss functions that have some relation to Monte Carlo integration and that we can use for optimization here.

We didn't really explore this topic in more detail, but it may be a really interesting thing to look into in the future.

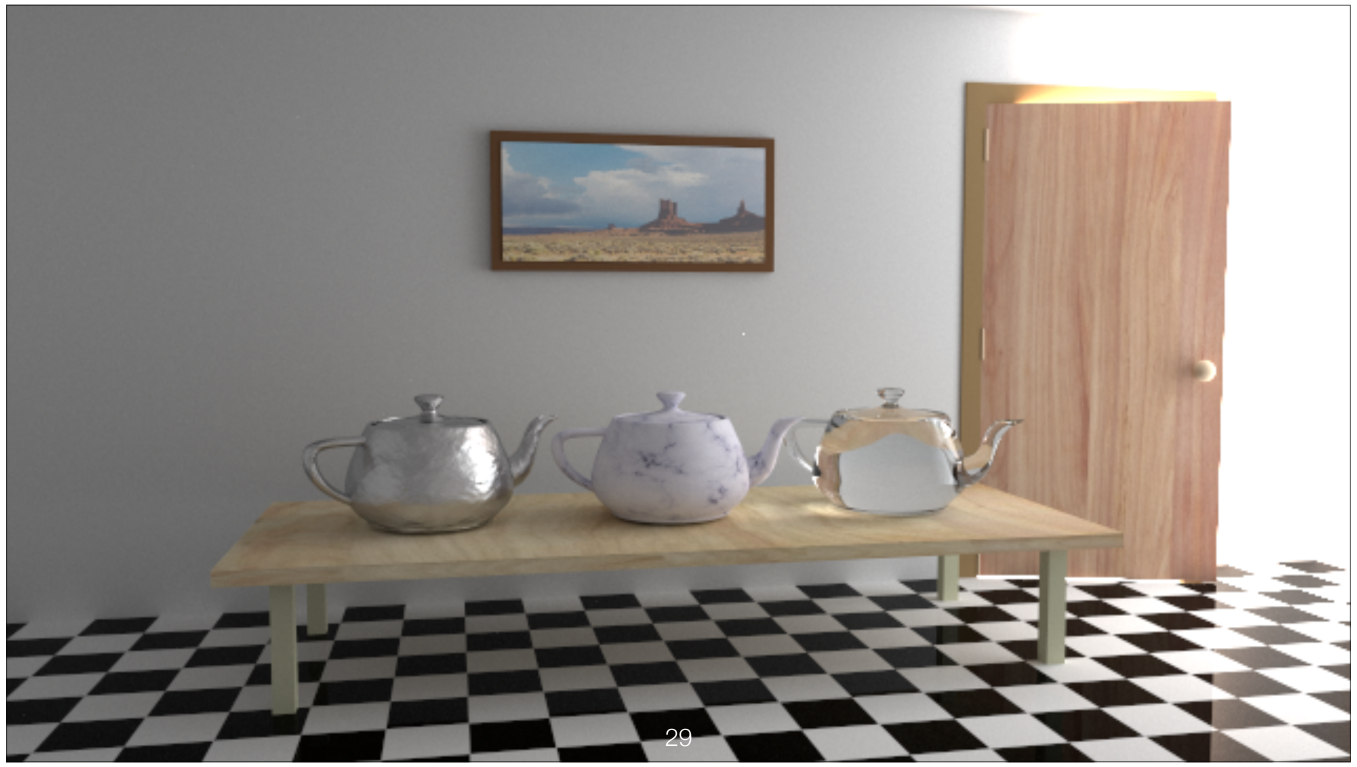
In practice we simply stuck with the KL-divergence for optimizing our networks because it gave the most robust results.

Putting it together...

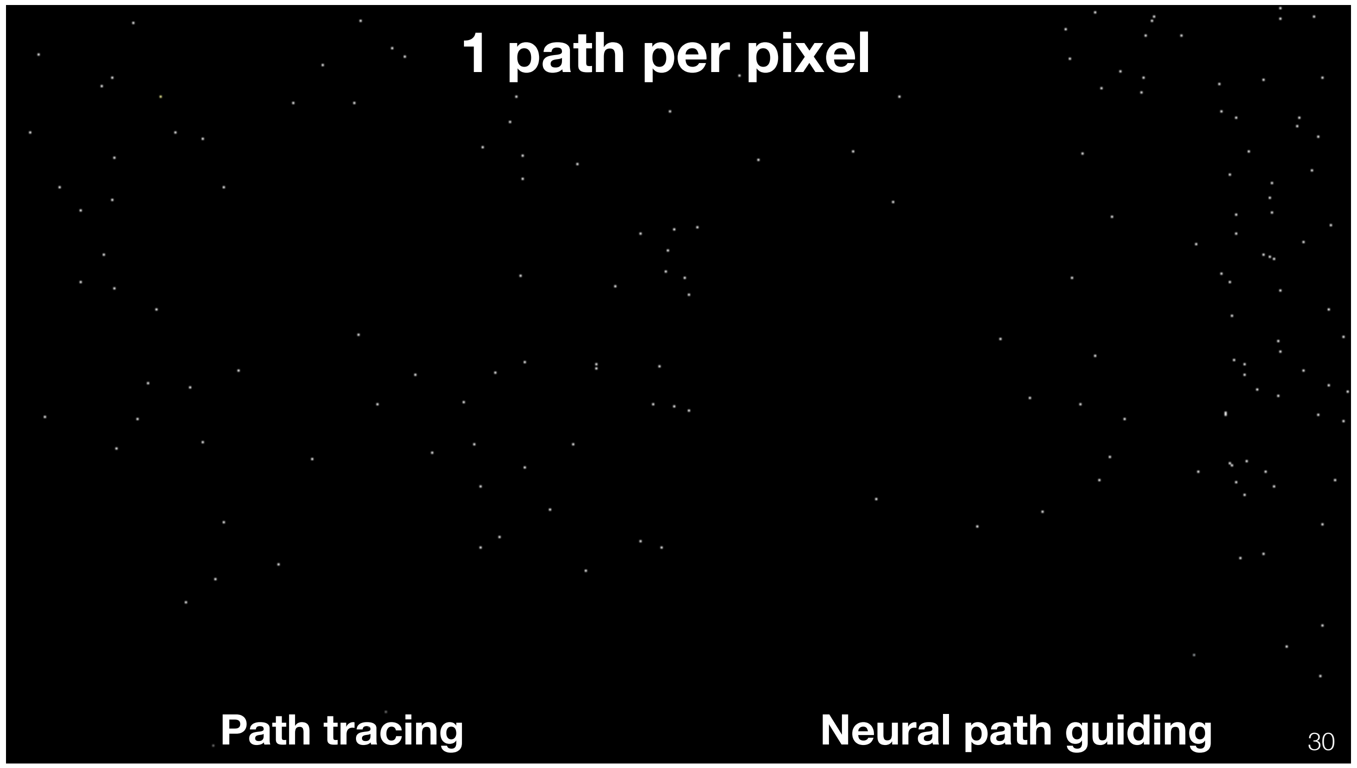


Alright, now you have a rough idea of how everything fits together.

Let me show you how rendering with this algorithm actually looks like.



We're going to render this image where all the light comes in through the door opening and we're going to compare what a regular path tracer does versus the neural approach.

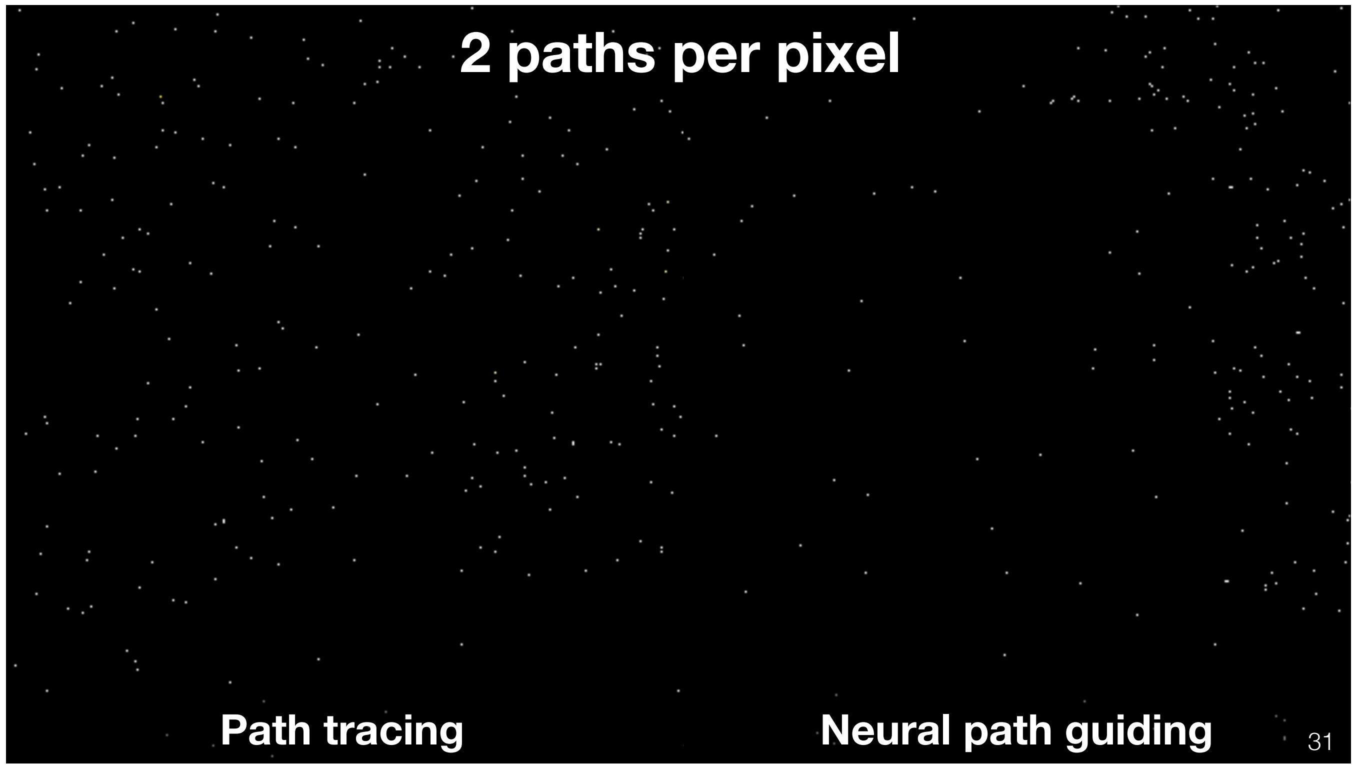


1 path per pixel

We start with an image where a single path is traced per pixel.

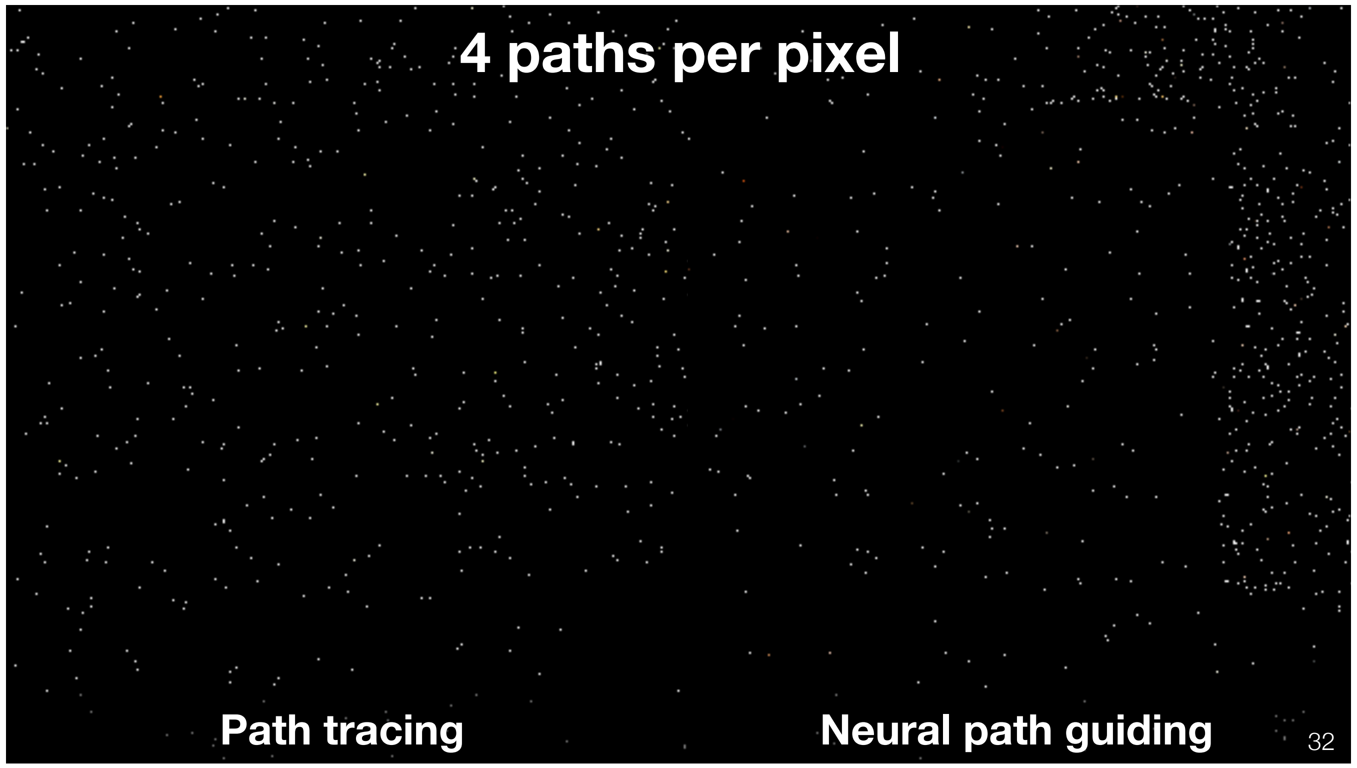
The left half of the image uses regular path tracing, whereas the right half uses our neural path guiding.

Since there wasn't any time for learning yet, both approaches have a really small chance of finding the door opening, so most pixels are black.



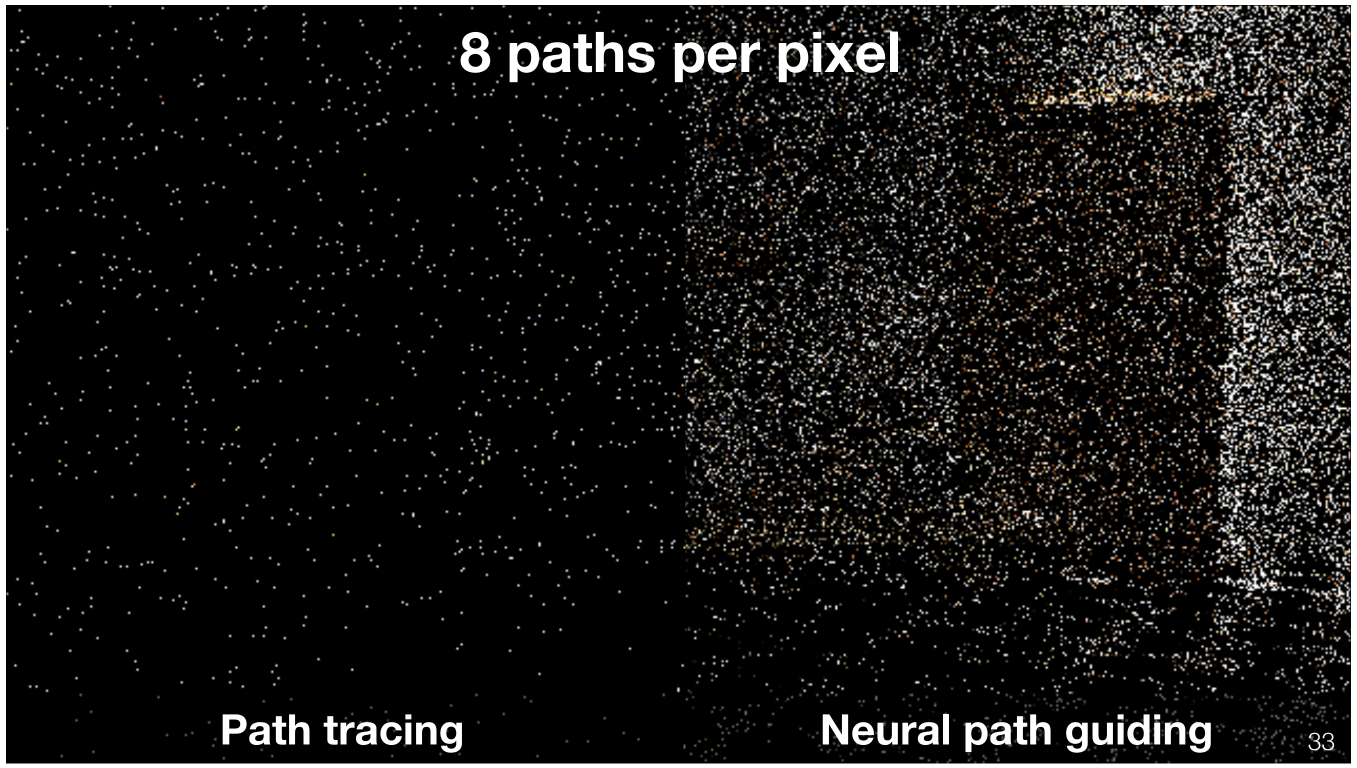
With two paths per pixel there is still not much of an improvement, so let's keep going...

4 paths per pixel

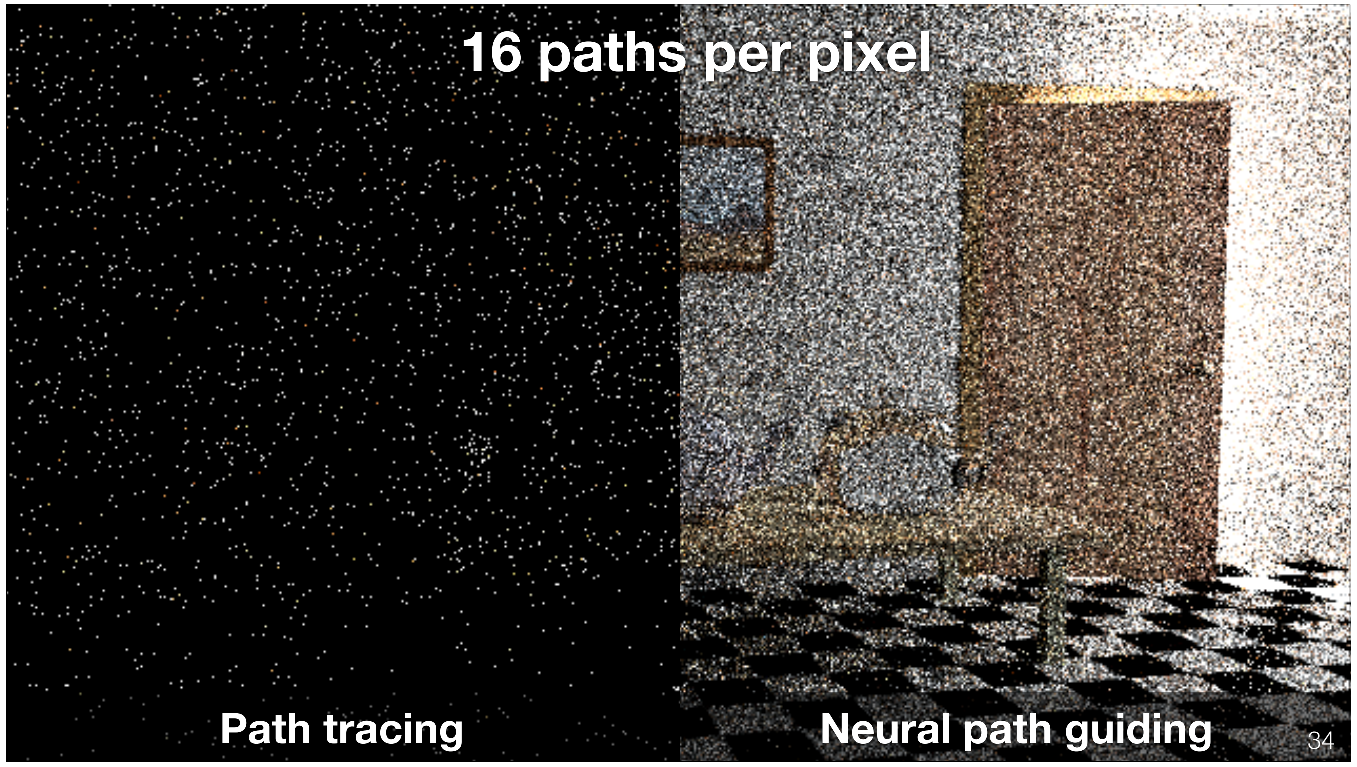


Path tracing

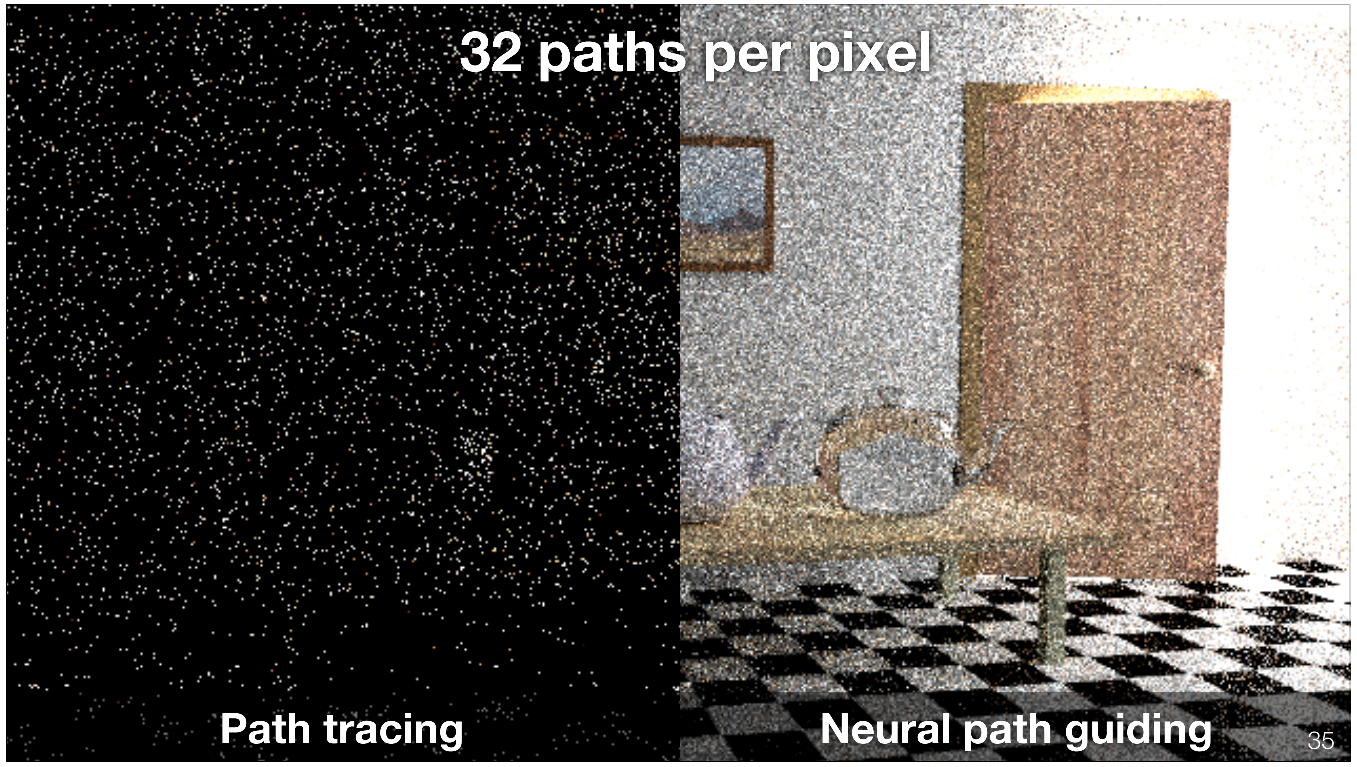
Neural path guiding

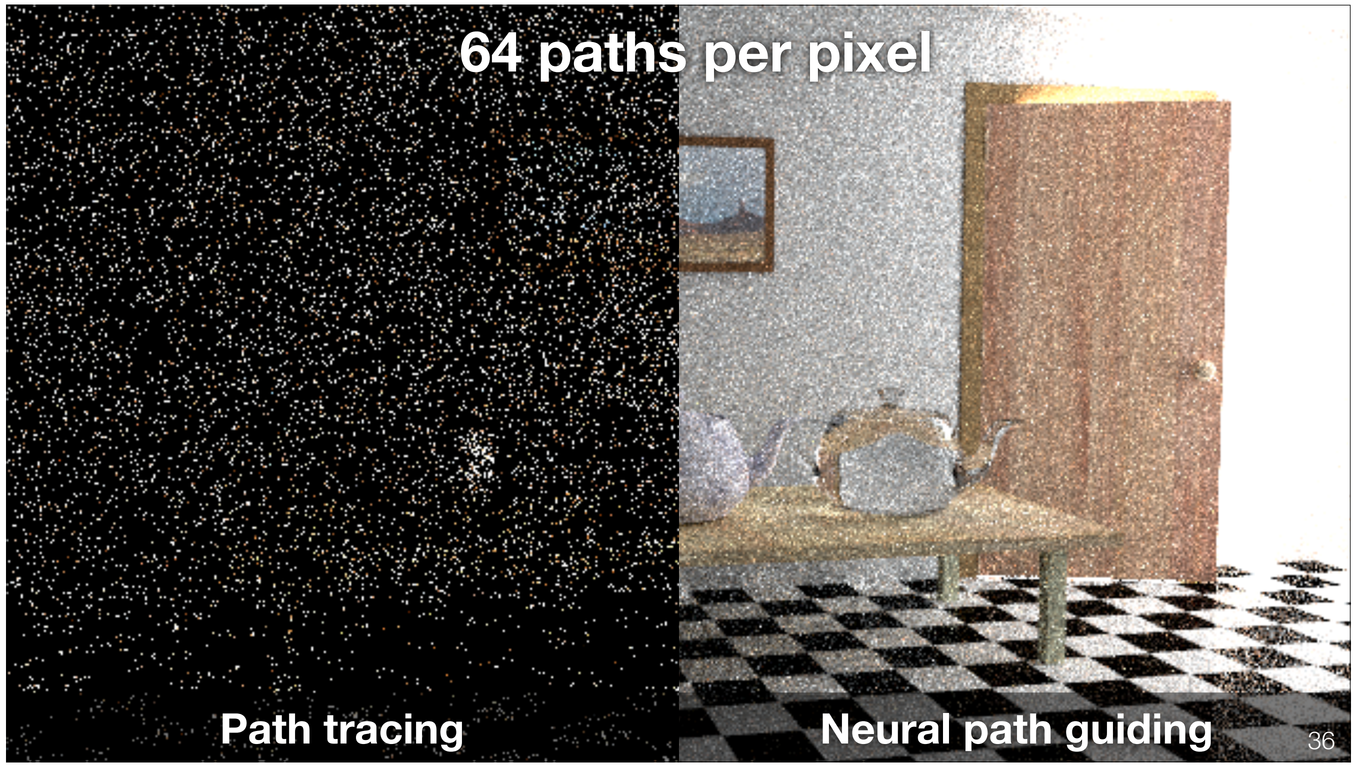


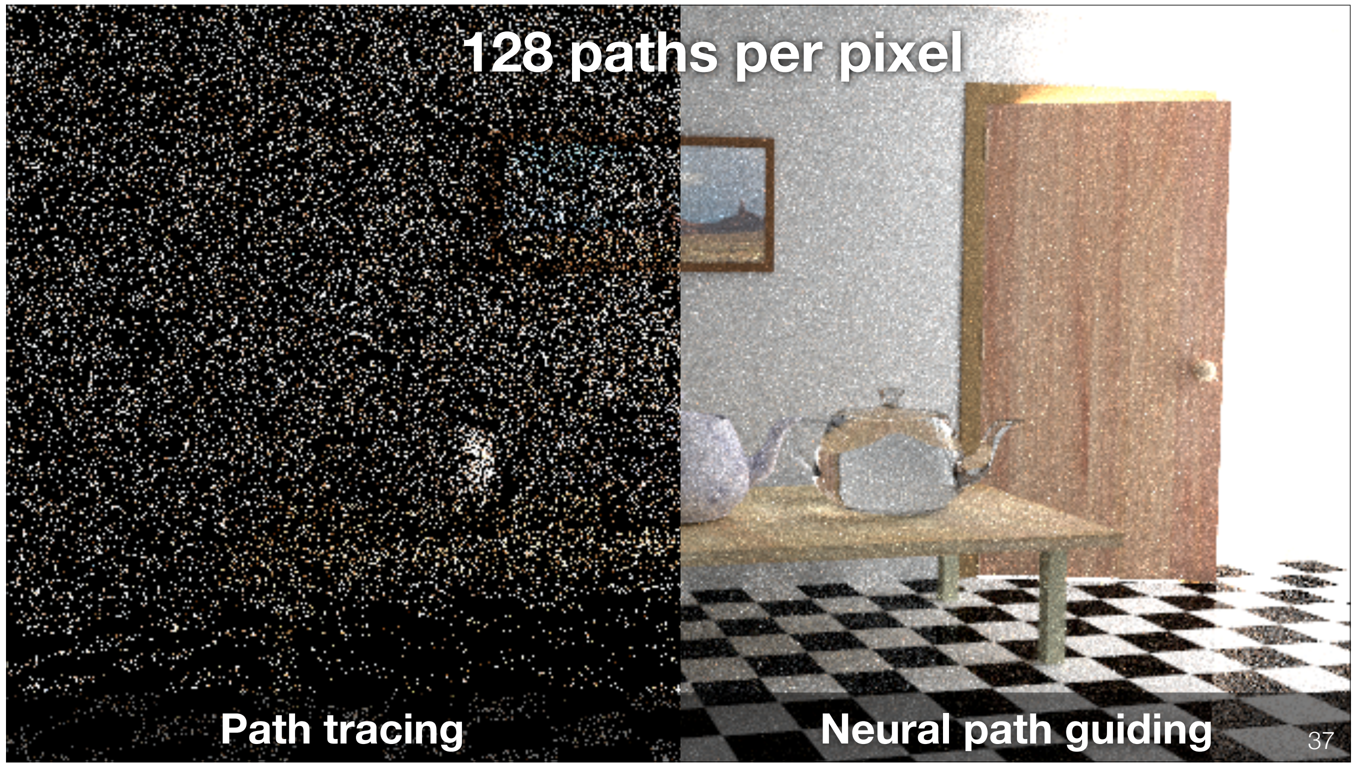
But after 8 paths per pixel, we start seeing the networks already getting much more paths right.



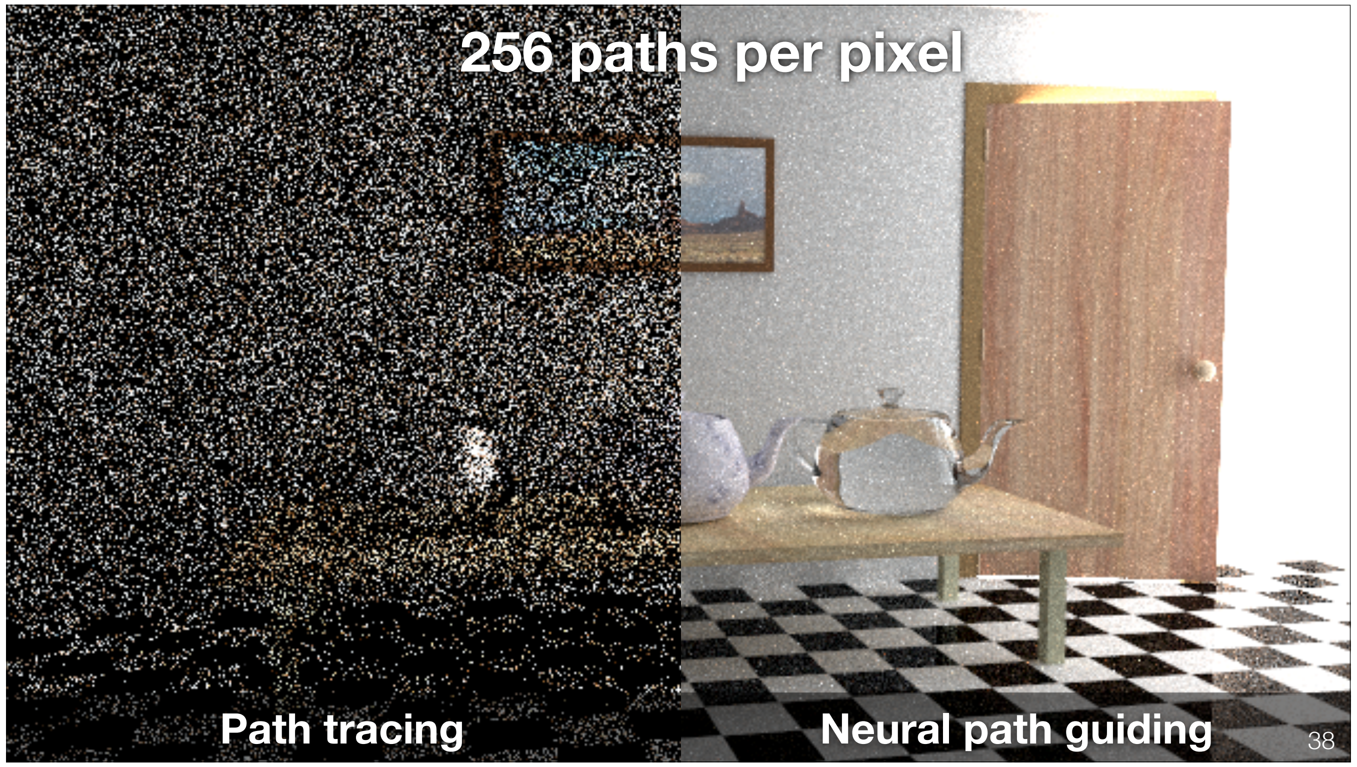
At 16 paths per pixel, the difference is getting really pronounced.

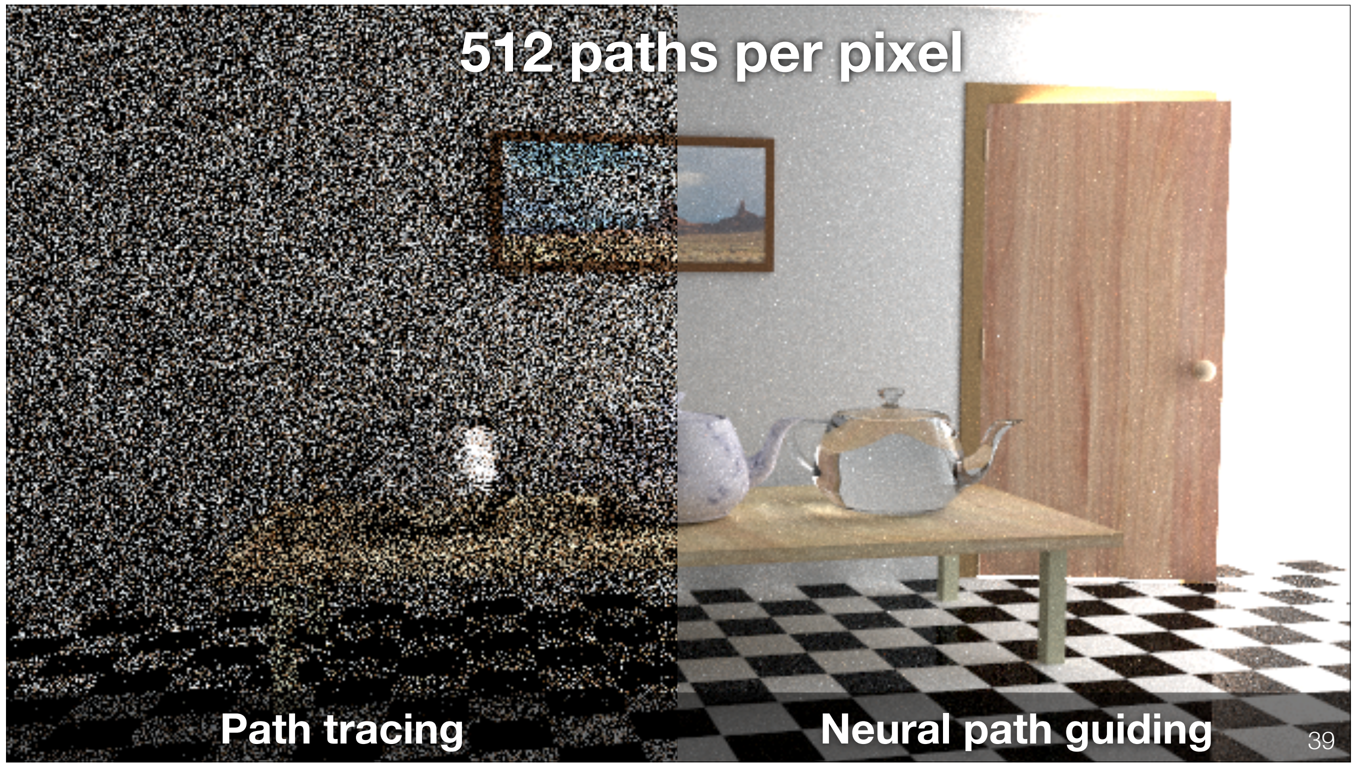

32 paths per pixel
Path tracing
Neural path guiding


64 paths per pixel
Path tracing
Neural path guiding


128 paths per pixel
Path tracing
Neural path guiding

256 paths per pixel

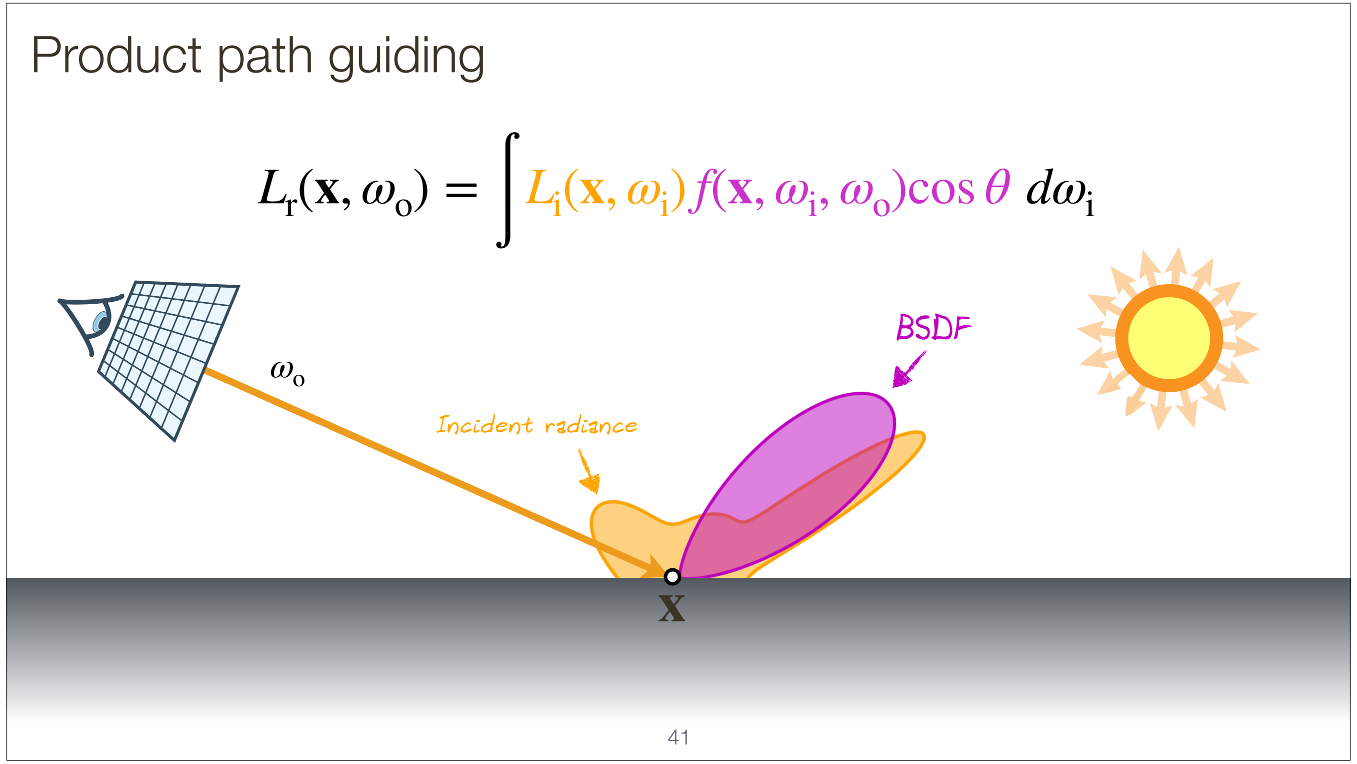




And this is what we end up with after 512 paths per pixel.

Product guiding

Okay, now let me show you some nice extra things that the neural networks allow us to do. For example, we can do product guiding!



Product path guiding

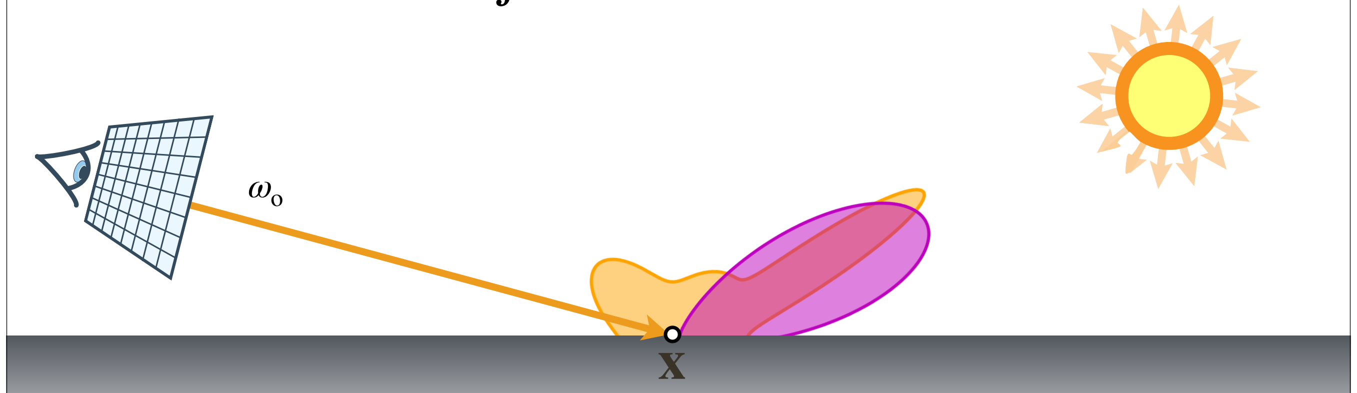
$$L_r(\mathbf{x}, \omega_o) = \int L_i(\mathbf{x}, \omega_i) f(\mathbf{x}, \omega_i, \omega_o) \cos\theta \, d\omega_i$$

Let's remember that the integral we are trying to solve is the reflection equation. We are integrating over the *product* of not just the incident radiance L, but *also* the BSDF.

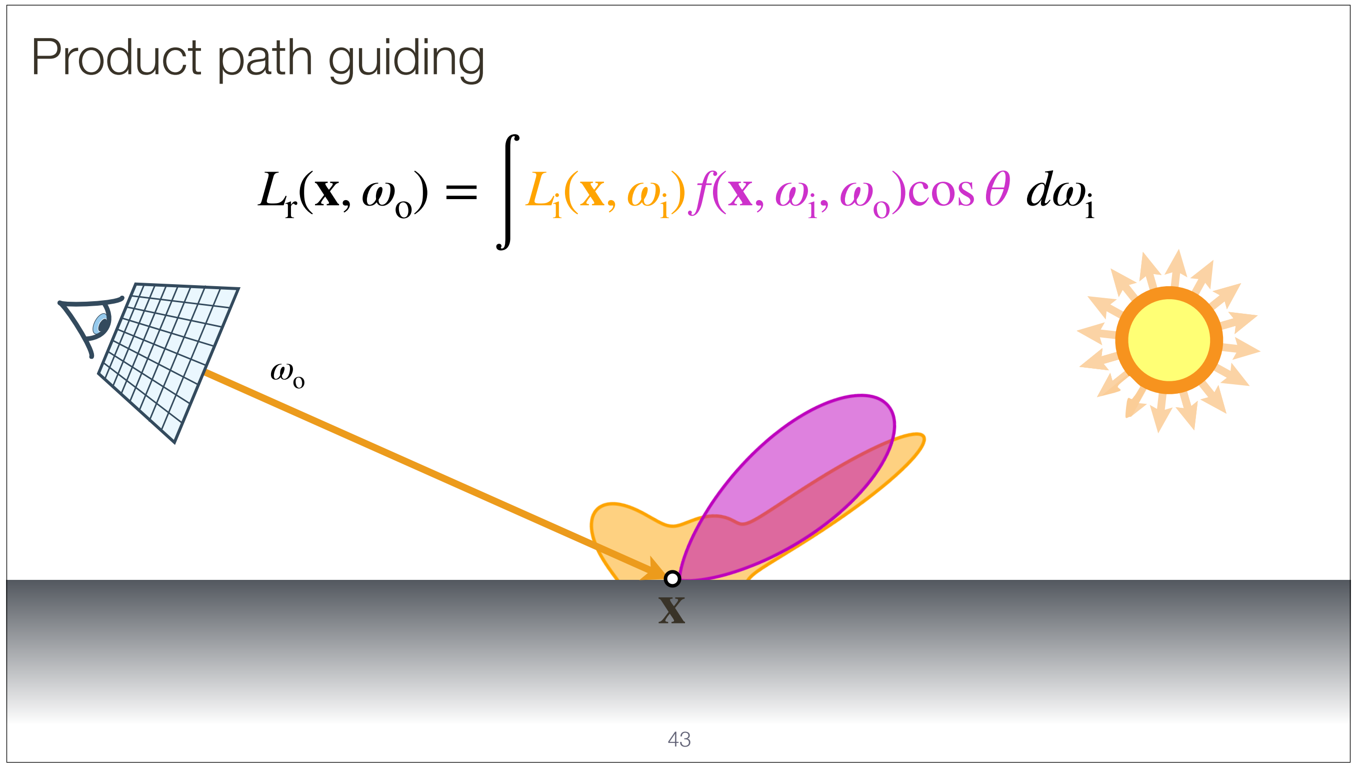
Previous path guiding approaches usually learned just the incident radiance, the reason being extra dimensionality. For instance, the incident radiance does *not* depend on where the material is viewed from, but the BSDF *does*.

Product path guiding

$$L_r(\mathbf{x}, \omega_o) = \int L_i(\mathbf{x}, \omega_i) f(\mathbf{x}, \omega_i, \omega_o) \cos\theta \, d\omega_i$$

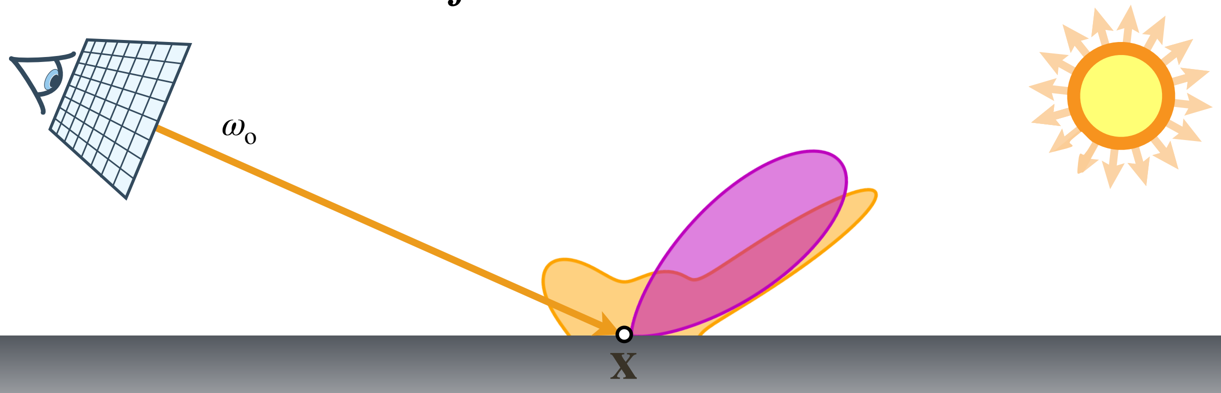


Like this.

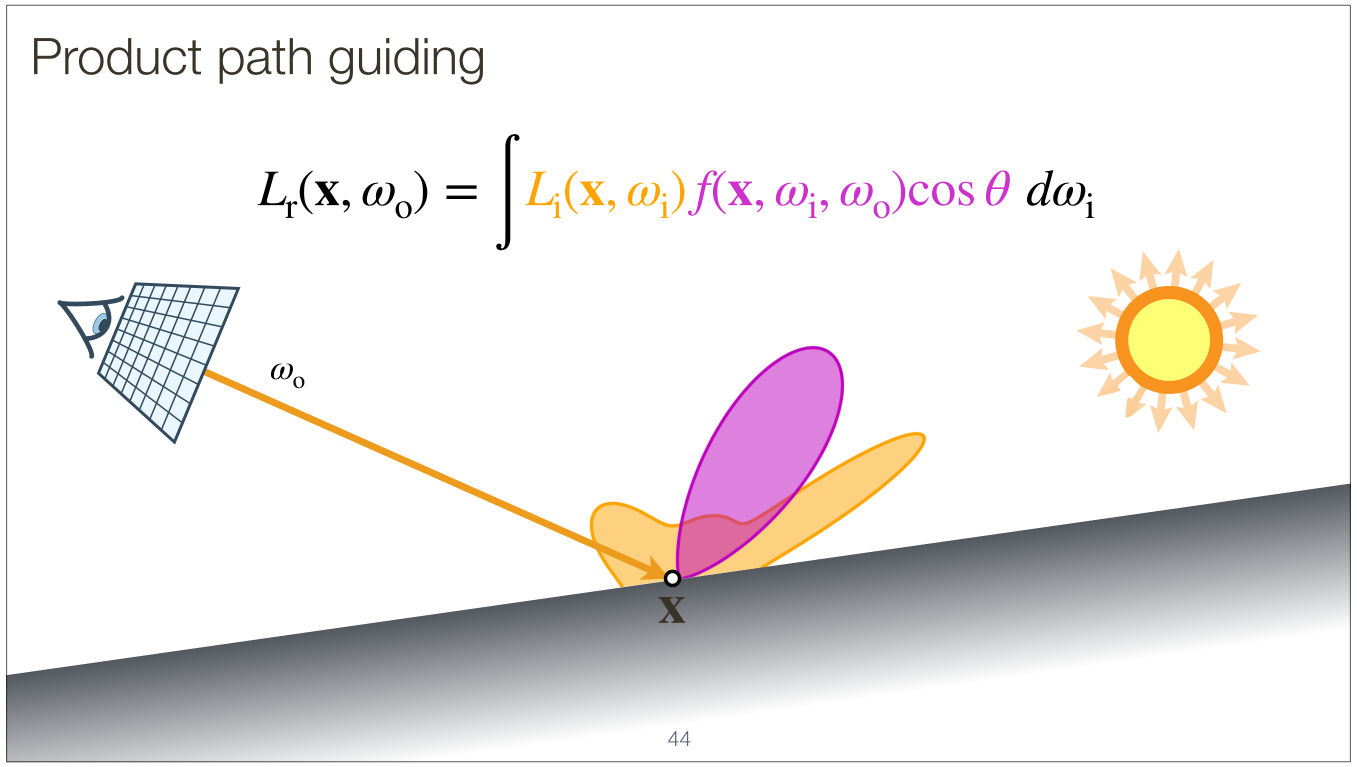


Product path guiding

$$L_r(\mathbf{x}, \omega_o) = \int L_i(\mathbf{x}, \omega_i) f(\mathbf{x}, \omega_i, \omega_o) \cos\theta \, d\omega_i$$



Furthermore, the BSDF terms often are highly dependent on local geometry; for instance, via the surface normal.



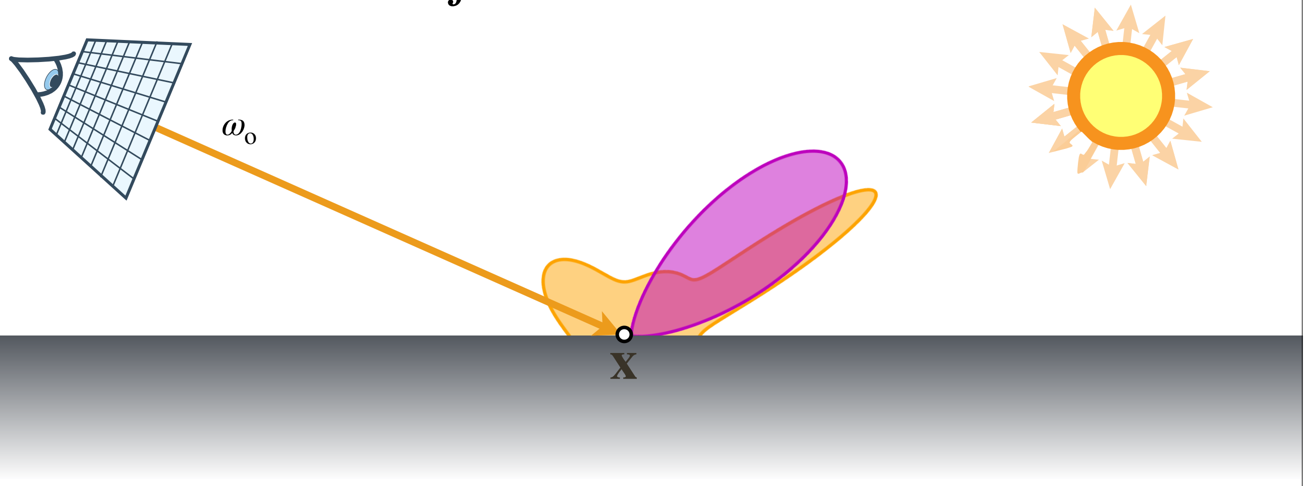
Product path guiding

$$L_r(\mathbf{x}, \omega_o) = \int L_i(\mathbf{x}, \omega_i) f(\mathbf{x}, \omega_i, \omega_o) \cos\theta \, d\omega_i$$

As you can see when I am tilting the ground.

Product path guiding

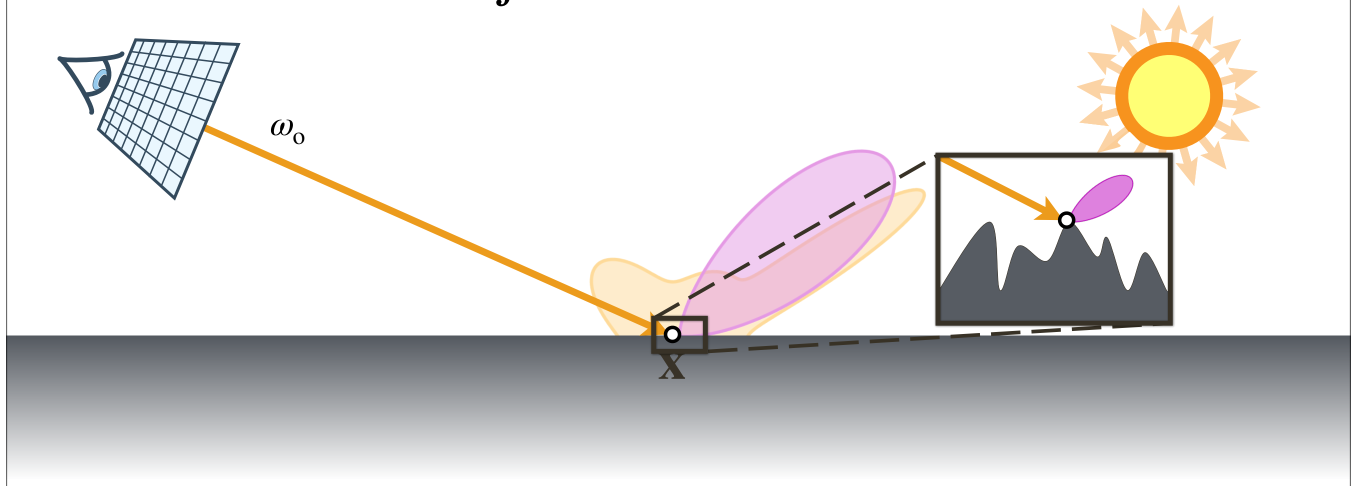
$$L_r(\mathbf{x}, \omega_o) = \int L_i(\mathbf{x}, \omega_i) f(\mathbf{x}, \omega_i, \omega_o) \cos\theta \, d\omega_i$$



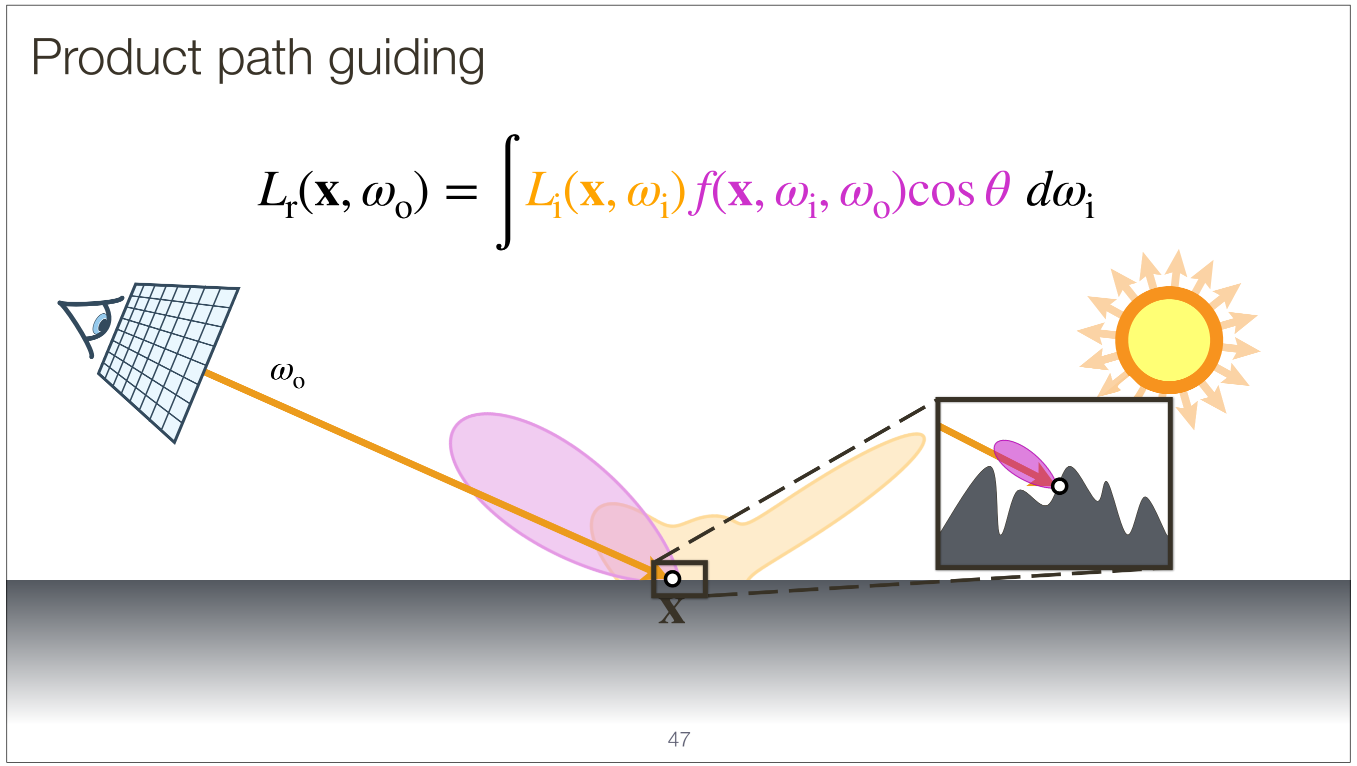
And this can make the problem far harder, because high-frequency detail can be introduced by textured parameters. Consider a bump map.

Product path guiding

$$L_r(\mathbf{x}, \omega_o) = \int L_i(\mathbf{x}, \omega_i) f(\mathbf{x}, \omega_i, \omega_o) \cos\theta \, d\omega_i$$

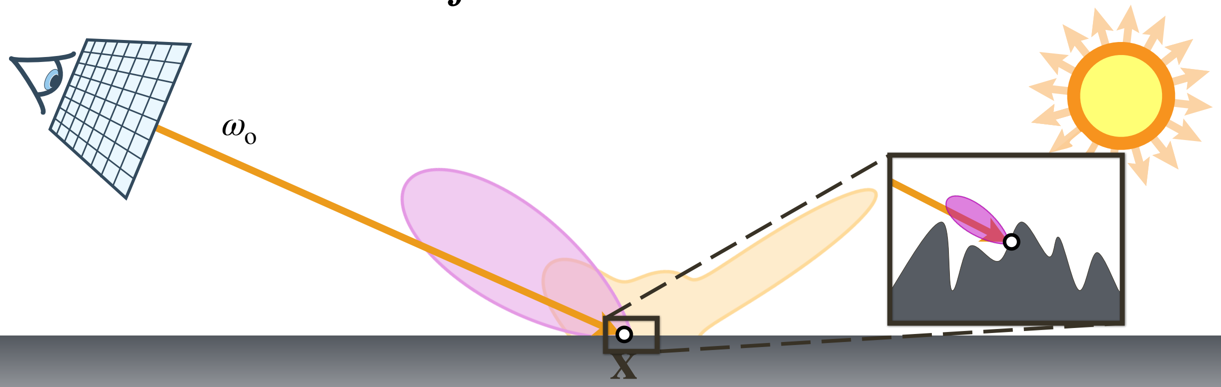


Here I am zooming in on our surface location to illustrate some small-scale variation in the normals via a bump map.

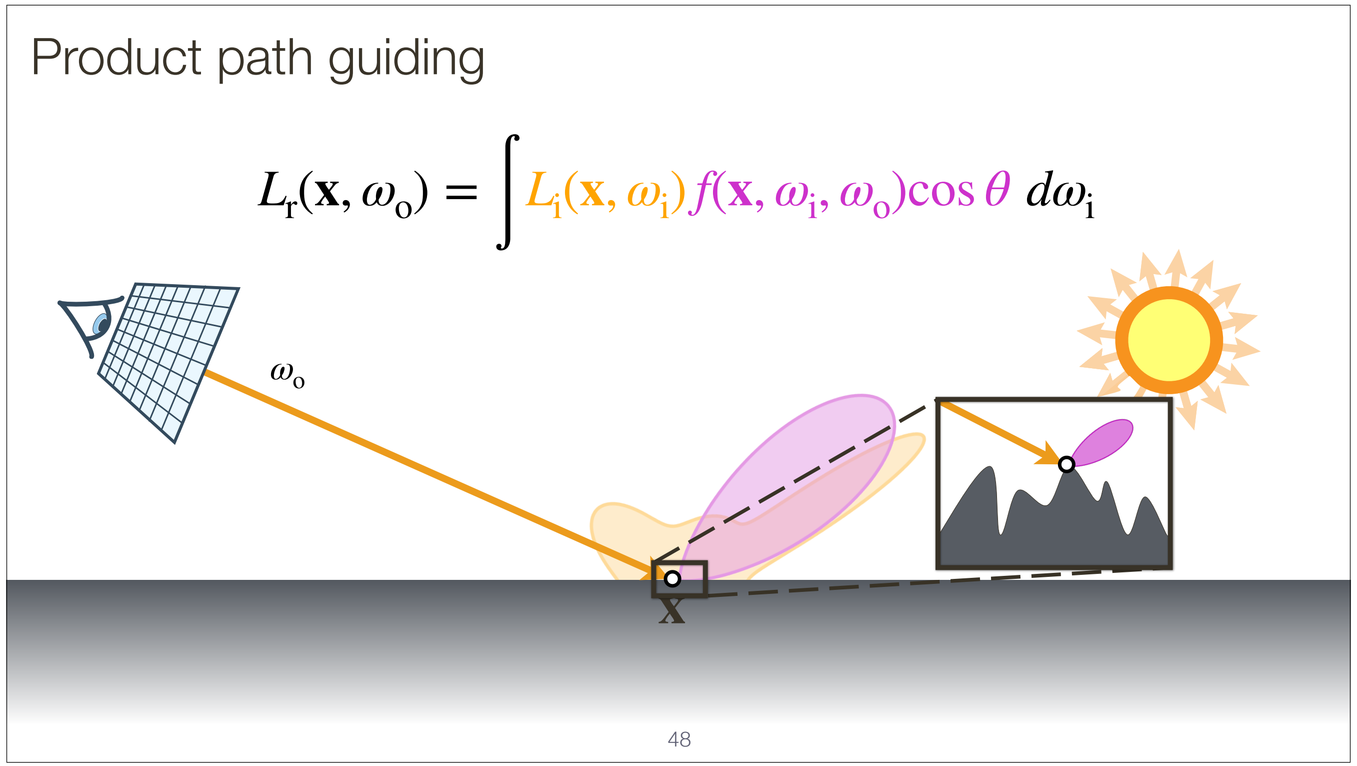


Product path guiding

$$L_r(\mathbf{x}, \omega_o) = \int L_i(\mathbf{x}, \omega_i) f(\mathbf{x}, \omega_i, \omega_o) \cos\theta \, d\omega_i$$



Moving our position just a *tiny* bit creates large variation in the product distribution, but not in the incident radiance alone.



Product path guiding

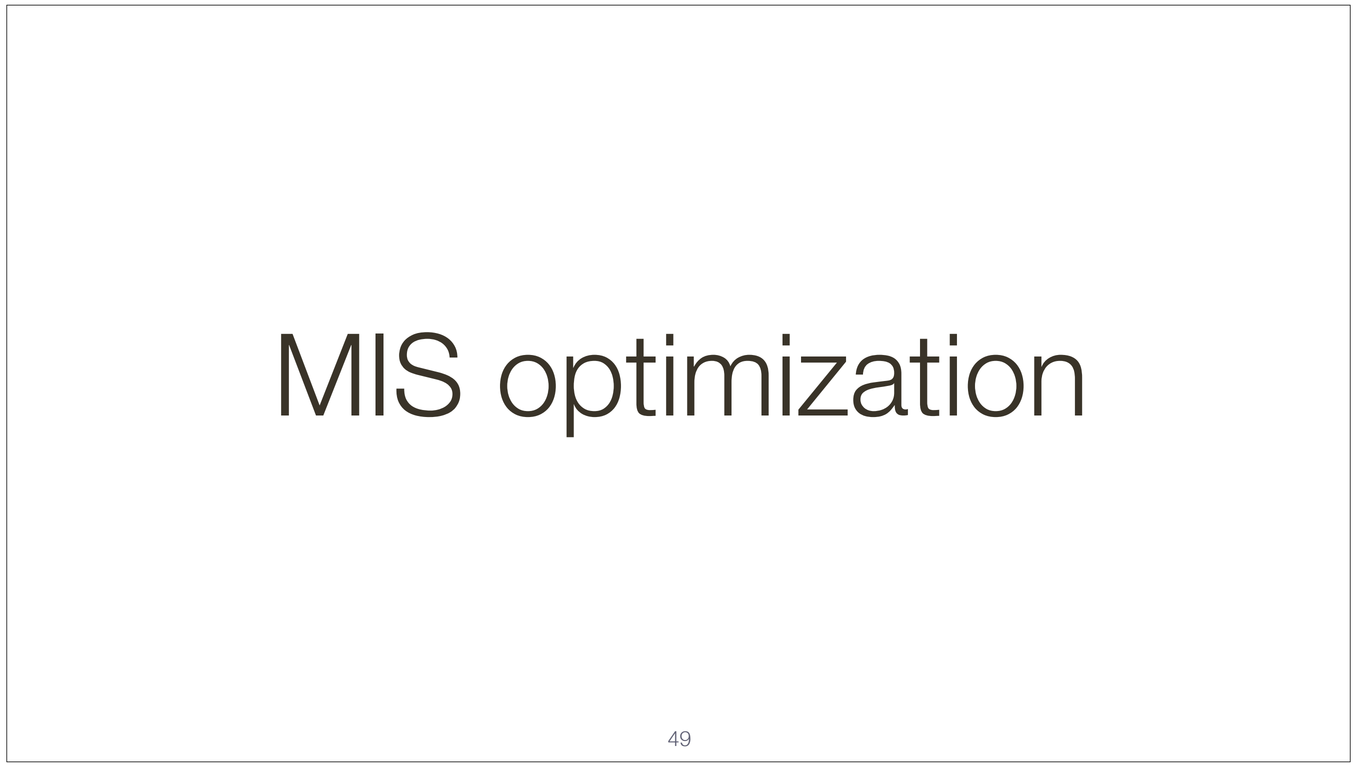
$$L_r(\mathbf{x}, \omega_o) = \int L_i(\mathbf{x}, \omega_i) f(\mathbf{x}, \omega_i, \omega_o) \cos\theta \, d\omega_i$$

ω_o
$\mathbf{x}$

And because of this, previous approaches generally struggled to learn the product. Some approaches learned the incident radiance and built the product on the fly, such as the works by Herholz et al., but none really learned the product holistically.

Neural networks, on the other hand, are pretty good at dealing high-dimensional inputs. We can simply feed the outgoing direction **omega_o** to our neural networks to capture the higher dimensionality of the problem, and we can additionally pass things like the surface normal---similar to "features" in de-noising---to address the bump map example you just saw.

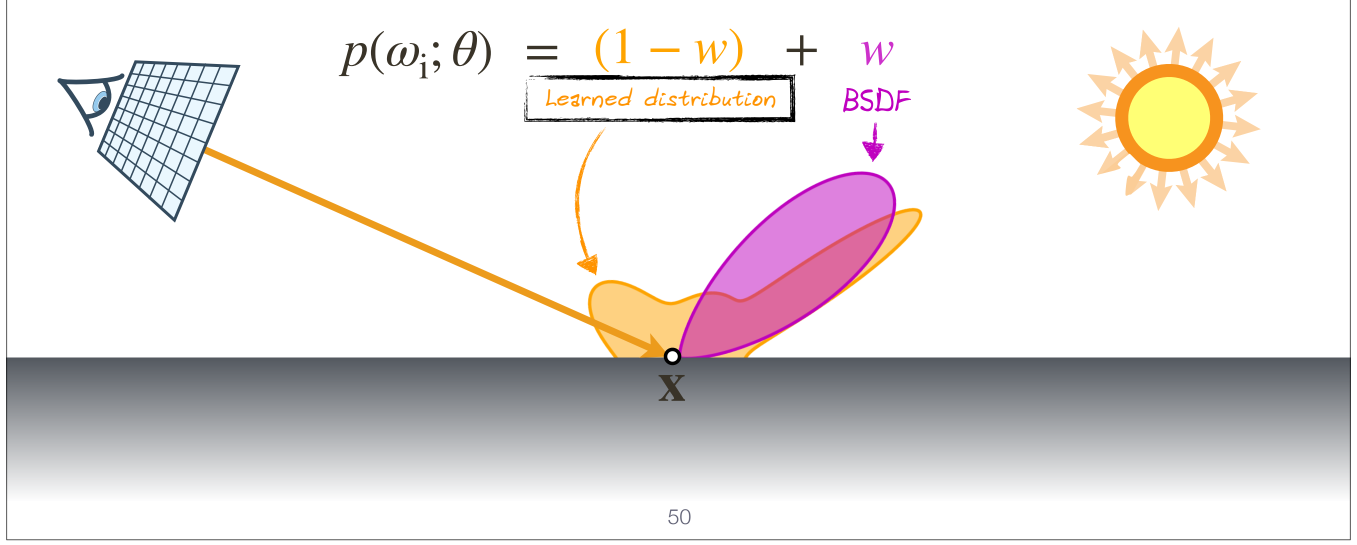
And with this, the networks can perform product sampling reasonably well.



MIS optimization

Another thing we can do with our neural networks is optimize how multiple importance sampling is done.

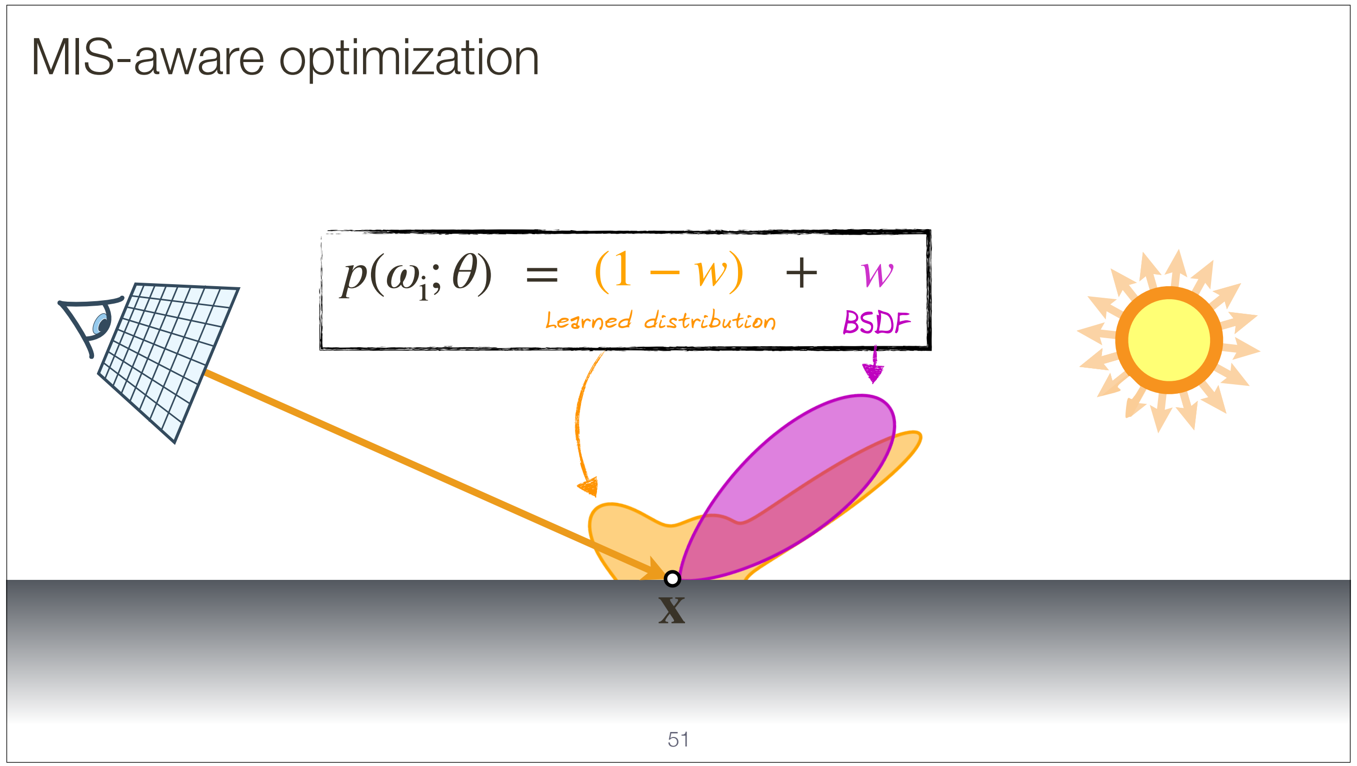
MIS-aware optimization



Usually, the learned distribution---in yellow---is combined with existing BSDF-sampling techniques via a process called the one-sample multiple-importance-sampling model.

This amounts to linearly blending between the two distributions. The approach I presented so far optimizes the learned distribution---the yellow guy---to match the target distribution---whether it is incident radiance or the product---as closely as possible. But if the final distribution we draw samples from is *actually* a multiple importance sampling *blend* between the learned distribution and the BSDF, why should we limit ourselves by trying to fit just the yellow guy?

Why not minimize the divergence between the *whole combined distribution* thing and the target?

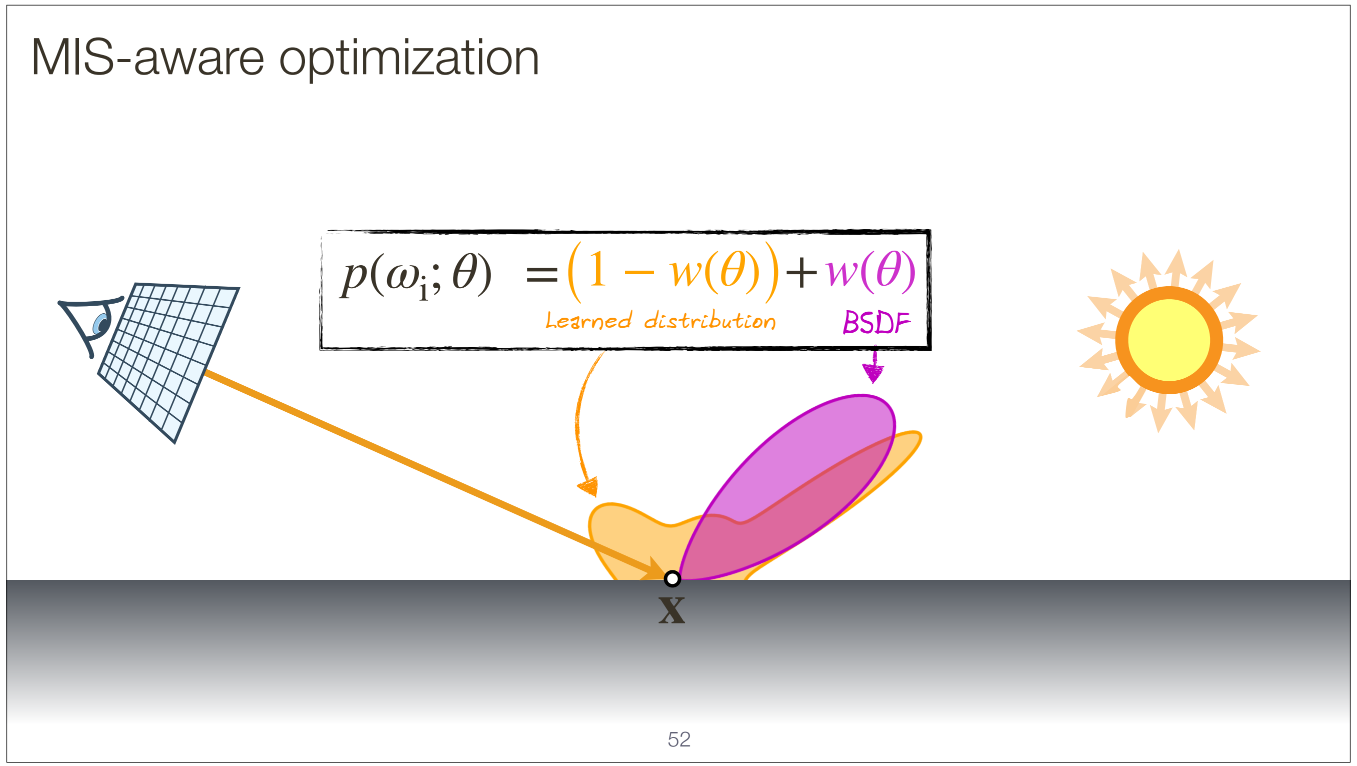


MIS-aware optimization

$$p(\omega_\mathrm{i};\theta) = \underbrace{(1-w)}_{\text{Learned distribution}} + \underbrace{w}_{\text{BSDF}}$$

With gradient descent, this can actually be done *trivially*! This has the effect of optimizing our networks in a multiple-importance-sampling-aware manner. If the BSDF already covers part of the integrand well, then our networks learn to not put additional samples there.

Now we're optimizing almost everything sampling-related... with the exception of the mixing weight **w**. Actually... let's also optimize that with yet another neural network!



MIS-aware optimization

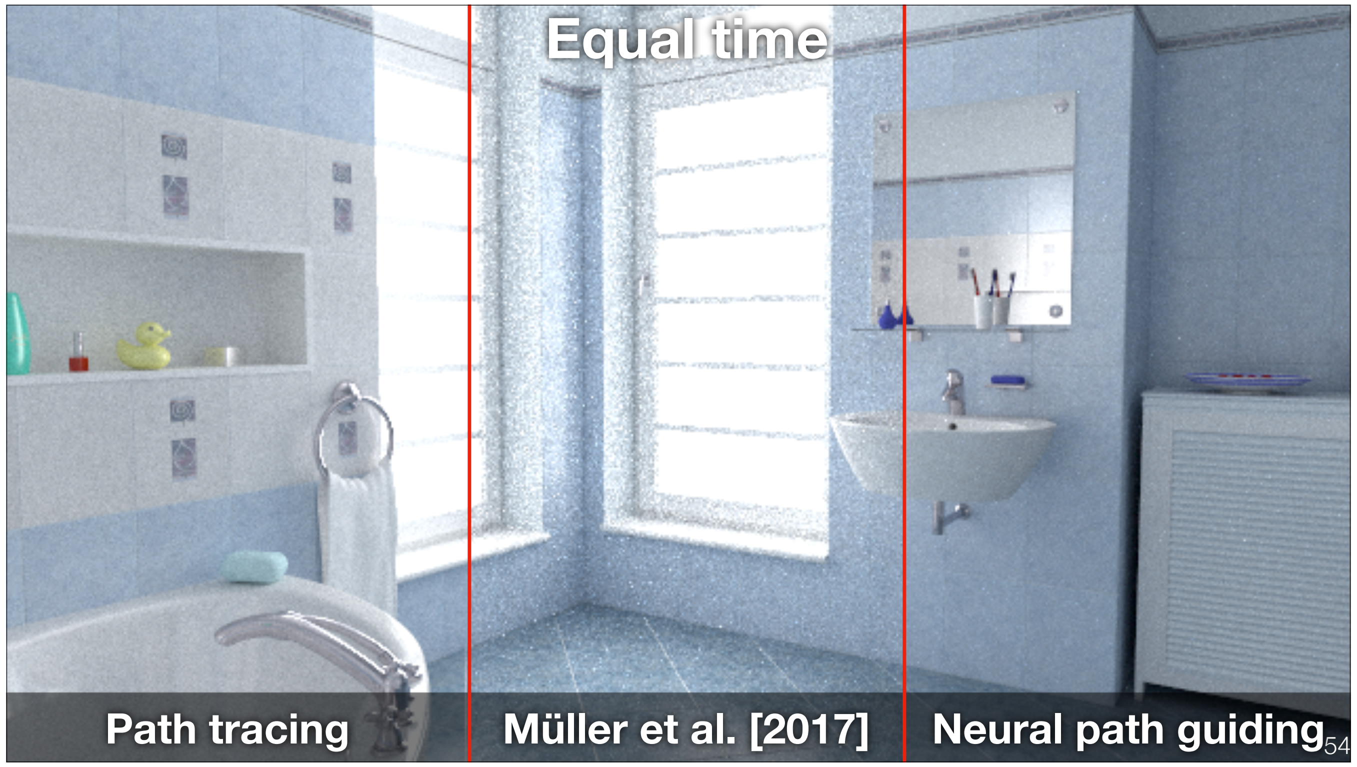
$$p(\omega_\mathrm{i};\theta) = \underbrace{\left(1 - w(\theta)\right)}_{\text{Learned distribution}} + \underbrace{w(\theta)}_{\text{BSDF}}$$

There we go. Now our neural networks have *full control* over the importance sampling pipeline, which means that---in theory---the can reach zero variance with infinite training data and model capacity.

[The following "bonus" slides demonstrate the improvement from MIS-aware optimization and some implementation details of our algorithm]

Results

Okay, those were the most interesting parts of our algorithm; now let's look at some results.

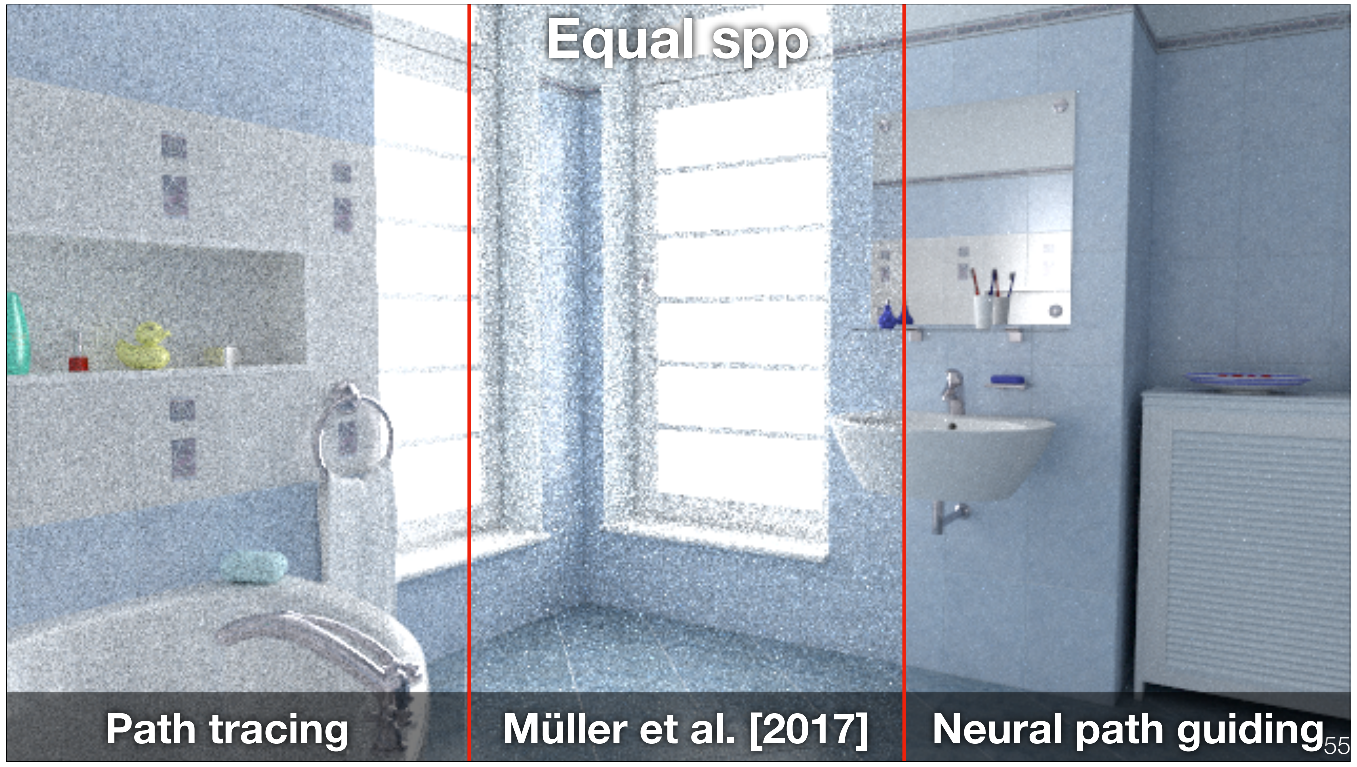


This bathroom is a relatively simple-to-render scene, which we use as a benchmark to make sure our guiding algorithm doesn't have adverse effects on previously simple-to-render cases.

...which is a problem that surprisingly many existing algorithms have, for example the path-guiding algorithm of Müller et al. The image actually becomes a little more noisy once that algorithm is enabled. Why is that? Well, that algorithm learns not to guide the product, but just incident radiance, and it turns out that the product is actually pretty relevant in this scene.

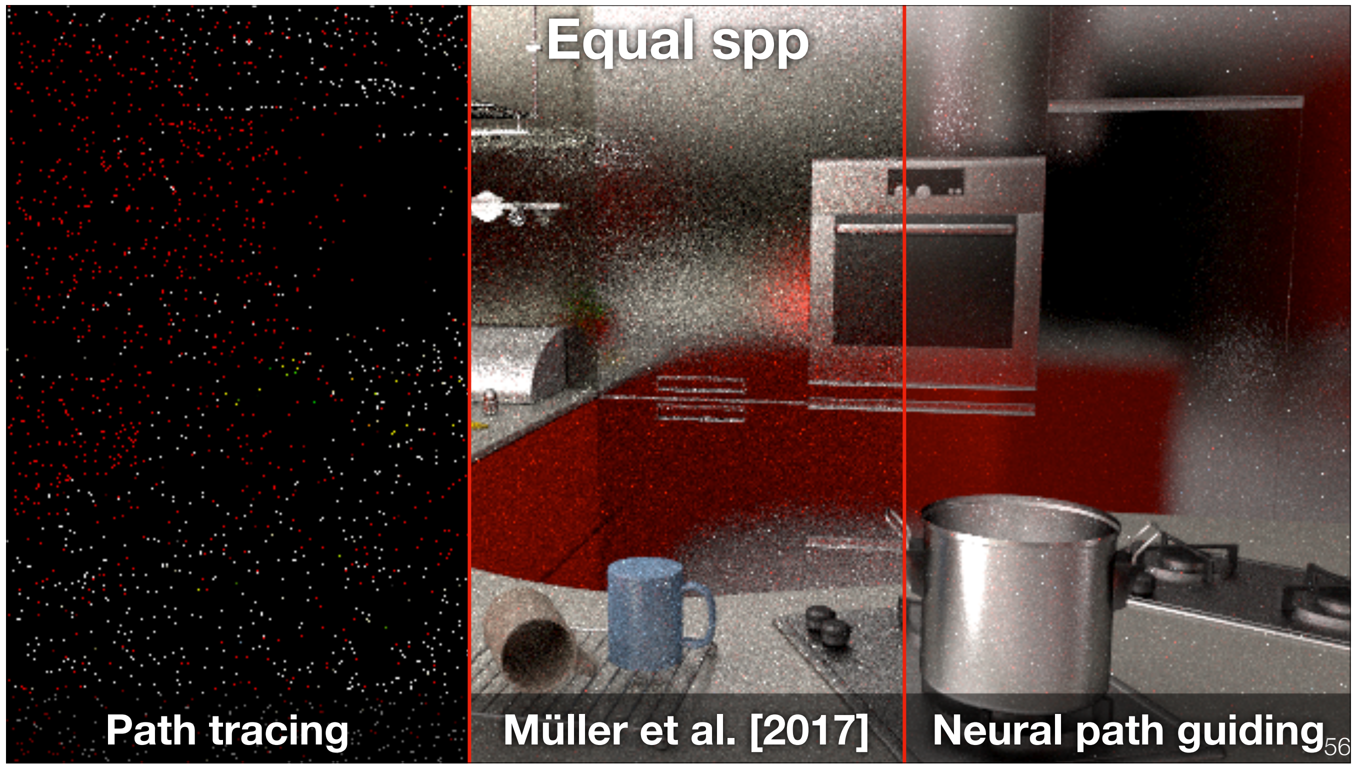
Our *neural* approach, because it learns the product, gives quality that's on par with path tracing.

But, let me point out that this is an *equal-time* comparison. Our neural networks are actually quite slow, so in this example we actually get comparable noise to path tracing with only about a tenth of the samples. If we did an *equal-sample-count* comparison, it would look like this...



So this shows that if we're somehow able to speed up the neural network stuff, it may actually be a game-changer for path guiding!

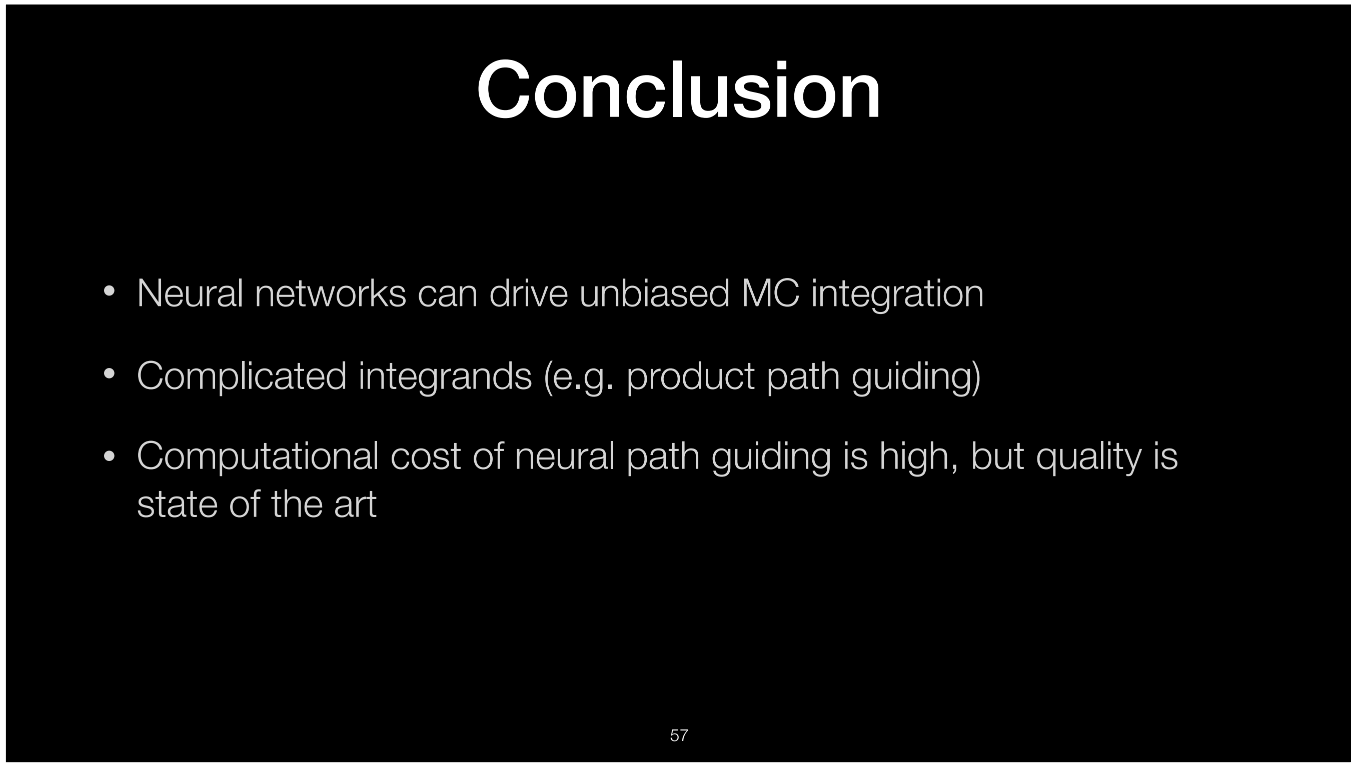
And such a speed-up is not too unrealistic of a hope---at this year's GTC, Alex Keller showed some initial results on linear-time algorithms for neural networks, which have the potential to be significantly faster than the quadratic-time algorithms we're currently stuck with.



Here's another result. In this scene path tracing is so bad, that probably only those people who worked with this scene in the past can actually tell what's going on from this mess of fireflies.

The path-guiding algorithm of Müller et al. improves upon this, but our neural path guiding goes quite a lot further.

Let me flip back and forth...



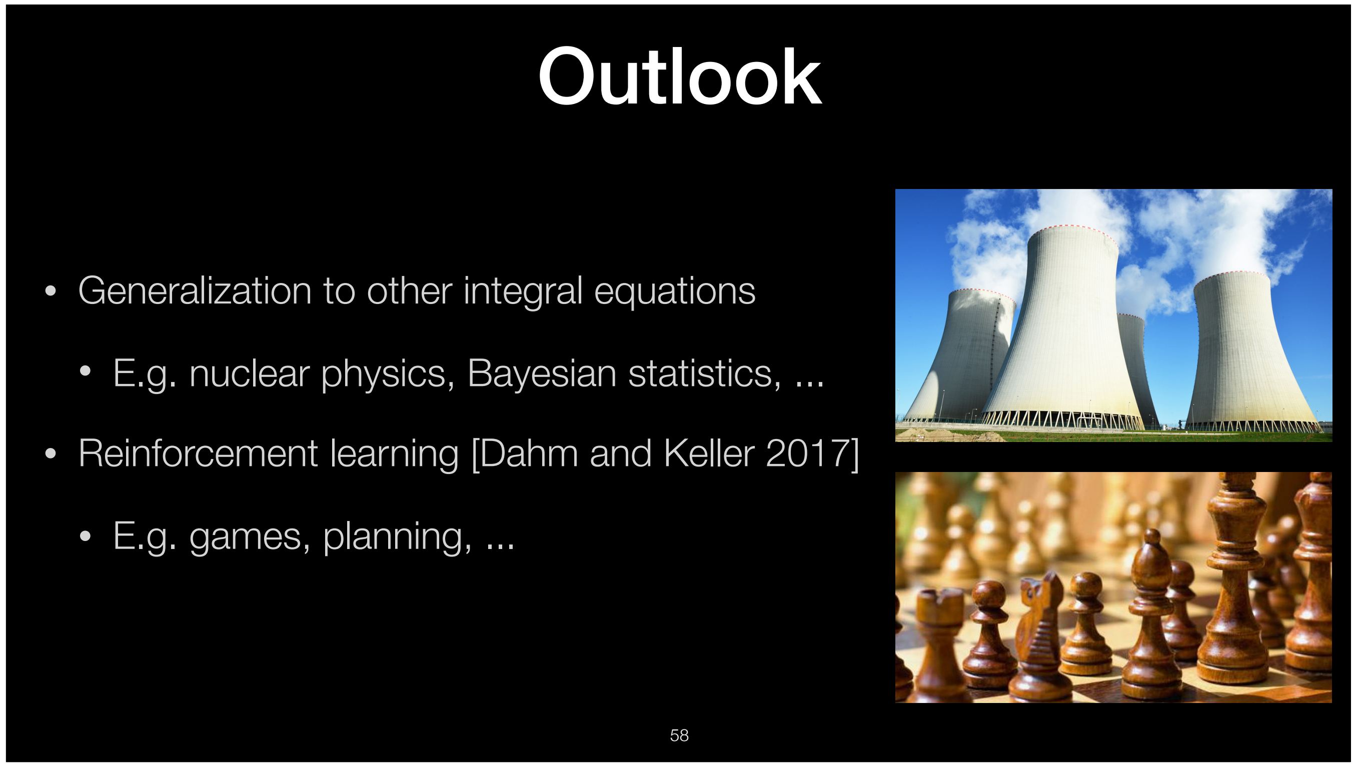
Conclusion

- Neural networks can drive unbiased MC integration
- Complicated integrands (e.g. product path guiding)
- Computational cost of neural path guiding is high, but quality is state of the art

So in conclusion, neural networks *can* drive unbiased MC integration... but they have to be applied in a very specific way. This kind of bridges the gap between the almost alchemical nature of hyper-parameter tuning and the principled, almost surgical, application of machine-learning to very specific problems.

Then, because neural networks are great at approximating complicated high-dimensional functions, we can use them for *really* complicated integrands, such as the product in the rendering equation.

Unfortunately, right now, the computational cost is pretty high so at least at the moment our approach isn't really practical. *But* its quality is state of the art, so there is hope for the future in case the performance can be improved.



Outlook

- Generalization to other integral equations
  - E.g. nuclear physics, Bayesian statistics, ...
- Reinforcement learning [Dahm and Keller 2017]
  - E.g. games, planning, ...

But what I think is most exciting in the future is to go *beyond* the horizon of just rendering.

Our neural importance sampling is actually much more general than its application to path tracing. It can, in theory, be used to solve a much broader class integral equations, which arise in nuclear physics, Bayesian statistics, and plenty of other important fields. In fact, we've recently been contacted by physicists from CERN who are interested in using our algorithm to predict high-energy particle collisions.

Another direction that is really interesting is reinforcement learning, because, as it turns out: the equations that underlie rendering are very similar to those that underlie reinforcement learning, which means that we may be able to apply our neural algorithms to problems like AI gameplay and other long-term planning tasks.

Thank you!

Interactive results

Co-authors
Brian McWilliams
Fabrice Rousselle
Jan Novák
Markus Gross

Misc
Abadi et al.
Dinh et al.
Thijs Vogels
Vorba et al.
Wenzel Jakob

Scenes by
Benedikt Bitterli
Jaakko Lehtinen
Jay-Artist
Johannes Hanika
Olesya Jakob
Ondřej Karlík
Magnus Wrenninge
Maurizzio Nitti
Marko Dabrović
Miika Aittala
Nacimus
Samuli Laine
SlykDrako
thecali
Tiziano Portenier
Wig42

With that, I thank you very much for your attention!