[SQUEAKING]

[RUSTLING]

[CLICKING]

JEREMY BERNSTEIN: OK, thank you, everyone, for coming to the lecture. So this was the example. Do people remember this from the homework? You remember? That's good. OK.

AUDIENCE: [INAUDIBLE]

[LAUGHTER]

JEREMY BERNSTEIN: It's a tricky one. Wait, OK, so if I just tell you, if I write, if I say that this is the gradient matrix. And it has what we call a reduced SVD. Do you remember what the answer was?

AUDIENCE: I think it's G G-transpose to get the sigma squared.

JEREMY BERNSTEIN: Oh. Yeah, something like that. OK, I'll just write it, or one version of the answer.

[INTERPOSING VOICES]

There's other ways to write the answer, but this is a simple way to write the answer. So you can think of this conceptually as, let's see, taking the SVD of the gradient. And you can think about this as setting all the singular values to 1.

And we call this a semiorthogonal matrix, because it's rectangular. It's not quite an orthogonal matrix. But I wanted to show you that there's someone on Twitter called kellerjordan0 who calls himself a speedrunner. So he wants to train the neural network as fast as he possibly can.

And he took like a variant of this method and then adds some tricks to it, like momentum. It makes it low precision. And then in order to compute, it's not really obvious how to compute this thing without doing an SVD.

And you don't really want to do an SVD when you're training a neural network because it's kind of slow. So he uses this iteration. So it's a way to compute this term without actually doing SVD. So it's just an iterative method to compute that thing.

And it throws away this learning rate, basically. It's a slight variant. But basically this is the result he's getting. So this blue curve is something called LLM.C, which Andrej Karpathy has a very fast C implementation of training transformer, which is supposed to be the fastest way to train a transformer.

But this guy Keller can train it in a fraction of the time using this special spectral gradient descent method. So the full disclosure is, I'm collaborating with Keller. So you take everything with a grain of salt.

But it seems to be actually much faster to use this method. And this all happened in the last couple of weeks, so actually after we wrote the homework. So just to say, we're trying to teach you some things that potentially matter in practice.

But anyway, the thing that we wanted to say is if anyone wants to look at this for a final project, it's an interesting problem of thinking about the math of optimization to actually make the optimization go faster. But anyway, that was just to tell you about that. That's not the main point of this lecture.

So last time, I think that you were learning about some aspects of representation learning. And there was this idea of developing techniques, contrastive learning techniques where you try to learn embeddings of your data. And you specifically want to take basically similar data points and map them to nearby embeddings.

And then you want to take a data point from a very different class and map it to an embedding that's far away. And we call these contrastive learning algorithms. And they're good ways to learn representations. Sometimes we think of them as somehow unsupervised ways to learn representations. But anyway, that was last lecture.

And then I also wanted to remind you that in even the earlier lectures, we were thinking about different perspectives on thinking about what's happening inside a neural network. And there was the idea of neural perspective, thinking about tensors, and then thinking about spectral properties of tensors.

So this is just recapping some different things. But in this lecture, not really using that, and just thinking more abstractly of a neural network as being a map from a sequence of vector spaces. So every layer of the neural network, you can think of it as a vector space.

And one possible goal of doing deep learning is to learn a really good representation of your data points. And I'm just visualizing that here as the idea of the input space, maybe all the different data points of unrelated things are all mixed up together and not really separated or disentangled.

And then when we do deep learning, we're hoping that as we move through the network, we get to the representation at the end where the data points from different classes or semantically different data points are separated from each other. And then we did a good job.

So that's what this is saying, that we want a representation of our data which is perhaps linearly separable at the end of the network. That would be nice.

So what I want to talk about in this lecture is the idea that when you pick what your neural network architecture is, it already has an opinion about which data points are similar to each other and which ones are dissimilar, even without doing contrastive learning. So I want to just introduce some tools for thinking about this.

And that's all this is saying, is that the neural architecture already expresses some kind of opinion about how similar two data points are, even before you train it. And we'll think about some tools to capture that idea. By the way, as ever, please just interrupt me if you have questions.

So the idea here is, I just give you some data. And I just want you to think about, just from first principles, ignoring everything you've ever learned in machine learning, about how you would actually fit it if you wanted to fit a model to this data. Does anyone have ideas?

AUDIENCE: Do a line, and then line curves down.

JEREMY BERNSTEIN: So I could somehow just interpolate maybe.

AUDIENCE: [INAUDIBLE]

JEREMY BERNSTEIN: That would kind of do that. Yeah, that's one idea, just linear interpolation. Yeah, I feel like-- Yeah, go ahead.

AUDIENCE: [INAUDIBLE] interpolation.

JEREMY BERNSTEIN: Polynomial interpolation, which is maybe harder to draw, but maybe I could fit some kind of quadratic to this. Potentially. If you just think about this, you can invent a lot of different function classes and just think about what it would mean to do machine learning with different function classes.

And it's kind of a big jump to get to doing neural networks, if you just think about things from scratch. But that's kind of where we are. So basically, there are a lot of classic approaches to doing machine learning, where people did just invent their favorite function class and say, hey, we're just going to do machine learning with this function class.

So a really classic example is called kernel methods. And actually it's quite related to what we talked about in the approximation theory lecture, which was maybe lecture two. But kernel method says we're going to take a kernel, which is basically, you could also think of it as a bump function.

And it could just look like a little Gaussian or a little curvy. And I'm going to place one kernel on the location of each input, if this makes sense. So if I've got four inputs, I'll put four little bump functions. And intuitively, if I've got four data points and I have four bump functions, then I have enough degrees of freedom to fit the data.

So that seems to make sense. And then I'm just going to rescale them with scalar coefficients just so that they interpolate the data. So it's four bump functions, four scalar coefficients, four data points. Everything checks out. And I wrote that down here.

So I construct my function, f of x, if I've got n data points, I need to sum over n bump functions. And I denote each bump function using this notation k. And then there's also a scalar alpha, which allows me to reweight the heights of the different bumps. And then if I have n bumps and n data points, I can definitely fit my data.

So it's kind of a natural way just to explain a little bit about what k of x and xi means. Roughly, looking at this, this would be k of x, 0. And then you should think about this xi as allowing me to translate this bump along the x-axis to be centered on that i-th data point.

That's just what this bit says. And then I can define a function space, let's call it big F, to be the set of functions f that can be decomposed in that form. And if I let the number of bump functions actually go to infinity, so I consider all possible locations of where I put the bumps, and then I actually allow myself to have infinite bumps.

Then, this is what is known as, they call this function space a reproducing kernel Hilbert space. And this was all the rage before deep learning or before the recent, let's say the last 13 years of deep learning. Before that, everyone was doing kernel methods, and they were considering spaces of functions like this.

Instead of choosing your neural architecture, the whole thing was like, you pick a kernel function. You choose a good kernel or a good bump function that somehow captures the structure in your data. And all the machine learning-- sorry, slight exaggeration, but there were a lot of machine learning papers focused on kernel methods.

So that's just a little introduction to kernel methods. And if you just think about it, it's not so obvious why do we use neural networks. Why didn't we go back to kernel methods? You need to think about it or it may not be obvious.

There's a second way, second classic way of constructing function spaces or doing machine learning, where actually this time, you think about sampling random functions. So you come up with some way of just sampling random functions. We would call this some kind of stochastic process.

And imagine you just keep sampling random functions from your stochastic process. And if it doesn't fit the data, you just throw it away. But if it fits the data, you actually keep it. And you're going to somehow end up with this distribution of functions that actually goes through your training data.

So if you actually use this method, you wouldn't actually do that procedure of sampling lots of functions. You would actually do something called conditioning the distribution of functions on the training data and getting what we would call the posterior distribution of functions. But intuitively, this is what's going on with this technique.

And there's a special way to make this computationally tractable. And it basically amounts to making the way that you generate the functions like a Gaussian thing. And then all the computations that you need to do this procedure become straightforward.

AUDIENCE: Question.

JEREMY BERNSTEIN: Yeah?

AUDIENCE: If I remember correctly, I think the optimal thing is roughly like a kernel method itself. How similar are they?

JEREMY BERNSTEIN: Yeah, that's a really good question. I have a slide. Actually, it turns out, as it happens, there's a close connection between this procedure and the thing on the previous slide with all the bump functions. And that's kind of interesting.

We've not discussed yet how to sample these random functions, but we're going to discuss them. But let's just say that we've discussed it already. And the point is that if you sample all of these random functions that are consistent with the data and then compute the mean of that distribution, you get a single, quite smooth function, which is the mean of that distribution over consistent functions.

And that turns out to be equivalent to the kernel interpolator of minimum kernel norm, for some definition of the RKHS norm. So yeah, there's a really close connection. But that should not make sense to everyone, because we haven't defined any of those things.

Oh, yeah. OK, here was the slide. So there's three ways now that we're going to think about function spaces. So first of all, this one is the one we just talked about where you sample lots of random functions, and then you throw away all the ones that don't fit the data, and you just keep the ones that fit the data.

And this will be something called a Gaussian process. Kernel methods are the ones with the bump functions, where we pick our favorite little bump and arrange them to fit the data. And then neural networks are what the class is mainly about.

And the cool thing, the really interesting thing is that there's all these correspondences. There's sort of a way to get a kernel method from a Gaussian process. There's kind of a way to get a Gaussian process-- they're related to each other. And the relation somehow goes both ways. That's all I want to say.

But interestingly, there's a way to get a Gaussian process from a neural network. Well, there's actually maybe a couple ways to do it. But the one that we're going to think about, there's a correspondence where a neural network in a particular limit is equivalent to a Gaussian process.

And it's basically when the network is infinitely wide. So you take the width of all the hidden layers and you make them infinitely wide. And then you randomly sample the weights. So you could think about that as you repeatedly, you just keep re-initializing your network. And it keeps giving you random functions.

If the network is infinitely wide, it turns out that this distribution is equivalent to something called a Gaussian process. So these connections, if you want to understand more theoretically about deep learning, it's interesting to explore these connections because you can relate neural networks to these classical machine learning methods that people really cared a lot about.

Then, you can ask things like, how useful are these correspondences? Do the correspondences actually amount to things which happen in practice? And that's something else you've got to think about. But I'm just going to tell you a little bit about one of the correspondences. Yeah?

AUDIENCE: These are all equally expressive, right? Or is there some sense in which one is more expressive than the other?

JEREMY BERNSTEIN: That's a good question. So the question was, is there a sense in which one is more expressive than the other? Or all the different function spaces equally expressive? I think they may just be different in some sense. You can look up and you'll find some literature that discusses this.

And it's things like the smoothness properties of the functions in the different function spaces may just be different. Like the kernel thing, because it's built out of bump functions, even though there's an infinite number of them, they may still be reasonably smooth functions.

But I think for a Gaussian process, depending on the covariate structure, you can have really horrendous functions. But interestingly, with that said, if you take the mean of the posterior of the Gaussian process, it's actually a nice, smooth function. So it's just different. I don't know which one is more expressive. But it's an interesting thing to think about.

OK, let's see. OK, so now I'm going to introduce slightly more formally what a Gaussian process actually is. So this is just a recap of what we just talked about. But basically, if we have some data, we can think about sampling lots of, lots of random functions that are the ones that are consistent with the data.

And we can get this distribution. And just notice that all the functions are going through the data, but they're random away from the data. And then given that random function, you can ask a couple of questions. One is, what does the mean look like?

And it's reasonable that the mean might be a lot smoother than all the random squiggly functions. And then you could ask, what is the standard deviation around the mean? And it's reasonable that it should grow as you move away from the data, and then shrink to 0 on the data. So that's one way it could look.

So the point of a Gaussian process is it gives you a formula, just a simple thing that you can easily code up. It just involves some matrices and some linear algebra. But it gives you a formula, both for the mean function and the standard deviation. In other words, given a set of data, you've got these inputs, you can pick a new point on the x-axis.

You say, I really care about this point on the x-axis. There's a formula that tells you the value of the mean and what the standard deviation is at that point. And then you can of sweep that point along the x-axis. And you can get this mean and you can get this standard deviation.

So it basically gives you a formula for drawing this picture. That's one way to think simply about what a Gaussian process is. And basically, the shape of these standard deviations and what this picture looks like depends upon something that we call the covariance structure of the Gaussian process.

And that's what we get to pick as machine learning people using Gaussian processes, is we get to pick the covariance structure. So that's the picture. Informally, I think this is probably the most useful-- if you just want a quick takeaway, you don't want to go into all the math, this is quite useful understanding of what a Gaussian Process is.

Just pretend that you don't know what a stochastic process is like. Most people honestly don't. But you do know what a random vector is. So let's just think about random vectors. So this is supposed to represent a vector, which I just turned horizontally so it would fit on the page.

And imagine that I have a multivariate normal random vector with some covariance matrix and some mean, which is actually 0. Then I could plot each coordinate. I get one sample of this random vector, and I just plot it on an axis.

So I just plot each entry. So this one corresponds to this one, this one corresponds to this one. And I could do that, and then I could just connect the dots. And I could call this somehow building a function from a random vector just by connecting the dots.

So I could do that if I wanted to. Then I could say, in this picture, it's not that great because neighboring points on the x-axis are not correlated with each other. And usually if I have a nice function, I want nearby points to be similar to each other.

So then you say, OK, hey, I'm just going to choose a covariance matrix with a special structure so that the covariance of neighboring entries in the vector are very correlated with each other. So I just pick the covariance such that neighboring entries in my vector are very correlated.

Obviously, if I permuted all the entries in the vector, it would break that. But I'm not going to do that. So I'm basically going to pick a covariance structure such that neighboring entries are related. And then I would get a picture more like this, where when I connect the dots, the function, it looks more like a nice function that I like.

And then basically a Gaussian process is exactly this construction, but then you let the number of grid cells in your vector, you let the dimension of your vector go to infinity. And basically, you pack these sample points more and more closely together, if that makes sense.

Does the construction make sense? You're saying if you have a random vector, you can plot its coordinates. And then you can let the number of coordinates go to infinity, and it will just limit to something that's more like a continuous function available.

Does that make sense? I've not really proved anything here. I've just given you an intuitive description. So if you forget everything else from the lecture, that's basically what a Gaussian process is. So now I want to give you a formal definition.

So now we say that we have some input space X, which that was our x-axis in the previous slide. And then for every point in this input space, for infinitely many points, we let f of x be a random variable.

So you think about the function f indexed by its argument x as being the continuous generalization of a random vector, where x is the index about which coordinate you are on on the vector. But that's just an informal thought.

And then what we say is that because it's difficult to think about infinite dimensional objects, we say that if we choose a finite number of input points and inspect what the random object f looks like on any finite set of inputs-- so on any collection of inputs x1 to xn, where n is finite-- we can consider that to be a random vector.

Legitimately, it is a random vector because we're inspecting the function on a finite set of inputs. And if this random vector is Gaussian for any finite collection of inputs, then we would say that f is a Gaussian process.

So this is the formal definition of the Gaussian process. It's kind of like an infinite dimensional object, where if you take any finite inspection of that object, it's a Gaussian vector, which is somehow related to what we were talking about on the previous slide. Yeah. Oh, yeah. Go ahead.

AUDIENCE: Given a random vector, how would you verify whether or not it is Gaussian?

JEREMY BERNSTEIN: It's more analytical. When we use this tool, it's more like you produce something which analytically, you know it satisfies this definition. You're very rarely in a situation where you would want to test empirically if it's a Gaussian. So you would have to verify it analytically in most cases. And I'm blanking on how exactly you would do that. But there are ways to do that.

AUDIENCE: So you can define in the construction what the random variables are? What each function [INAUDIBLE], right?

JEREMY BERNSTEIN: Yes. So if you're going to use Gaussian processes, you will build something which satisfies this definition.

AUDIENCE: OK.

JEREMY BERNSTEIN: And the thing is, what's difficult about this definition is it's not obvious that anything should satisfy this definition. And there's a theorem that most people who use Gaussian processes don't know.

But I think it's called Kolmogorov extension theorem or something like this. And there's this area of math which is about proving that these things exist. But it turns out you actually don't need to know that area of math to actually use this stuff.

AUDIENCE: And then I was just wondering, precisely what is the definition of a Gaussian vector? I can see from--

JEREMY BERNSTEIN: Ooh!

AUDIENCE: --the slide how you're just sampling from a Gaussian to construct it. But [INAUDIBLE].

JEREMY BERNSTEIN: Yes.

AUDIENCE: --want to know what it means.

JEREMY BERNSTEIN: The simplest Gaussian vector is a Gaussian vector where the coordinates are iid. And then just for each coordinate, you sample a different Gaussian. And they're not related to each other.

That's a simple one. And then a general one is defined with respect to a covariance matrix. And I'm probably going to butcher any explanation I give now. But I would just go on Wikipedia after the class.

Another way to think about a general multivariate Gaussian is to do the iid coordinates and then rotate the space. And it turns out that's equivalent. I'm pretty sure that's basically equivalent to a general one, with rescaling the coordinates as well. Yeah?

AUDIENCE: Is this just a fault of the dimensionality? Or if it's spaced within [INAUDIBLE] Gaussian lives would be the number of data points, right? [INAUDIBLE]?

JEREMY BERNSTEIN: Yeah, but for this definition, in this case, that would be if I had a finite thing. But this definition is for any collection of n for any n. So it has to hold for all possible collections of inputs. It's quite a technical definition. Sorry, is that clear?

AUDIENCE: Yeah, I guess just initially when I first thought Gaussian, I just imagined the Gaussian curves. So I imagine that being Gaussian. But obviously that's not what this is. And then I thought maybe [INAUDIBLE]. But it looks like--yeah. I understand better now.

JEREMY BERNSTEIN: Yeah, it's conditioned on having a set of inputs.

AUDIENCE: I'm thinking of a neural network, and it's defined on some data points. That's Gaussian in the sense that the function that it's applying on the data points in the dimension of the space, the dimensionality of the data points for, I guess, larger is Gaussian. I guess I'm just trying to reconcile this with the neural network not being--

JEREMY BERNSTEIN: Yeah.

AUDIENCE: [INAUDIBLE]

JEREMY BERNSTEIN: OK.

AUDIENCE: I'm thinking I'm close, but [INAUDIBLE].

JEREMY BERNSTEIN: I have a concrete example. I'm just not sure how much further into the lecture it is going to be. But I think it's going to resolve this question.

So let me just plow on, but keep the question. But I just have a few more high-level things to say. And then, oh, yeah, I have a picture as well. So let me just go on. I think I'm going to resolve your question.

So I should have defined what a standard multivariate Gaussian is. And I'll add that for next time. So this is just to say that a Gaussian process is the thing that generalizes a finite-dimensional Gaussian vector to somehow an infinite-dimensional random function.

And if a Gaussian random vector is described by a covariance matrix, well, actually this thing that we call a Gaussian process is going to be described by something that we call the covariance function, which tells us-- at two different inputs, what is the covariance between the function at those inputs. So that's how the covariance matrix gets generalized.

And an example, if in the finite case, the covariance was e to the i minus j squared, then here in the continuous case, it would be something like e to the x minus x prime squared. So that's how that would generalize. So hopefully this slide will help build some more intuition for what we're talking about. OK, go ahead.

AUDIENCE: [INAUDIBLE] dimensional function, you just need a function that accepts an infinite number of inputs?

JEREMY BERNSTEIN: Exactly.

AUDIENCE: OK.

JEREMY BERNSTEIN: Exactly. Let me just say again that, yeah, exactly, we've got the x-axis or the real line as our input space. So there's an infinite number of things on it. And here we're going to consider two examples, x very close to x prime, which is this picture.

And basically, we're going to choose the covariance function almost always in our Gaussian process, such that if I sample the random functions and I plot a scatter plot-- so I sample one random function and I get this point for f of x and f of x prime. And then I sample a different random function, and I get this point.

And I sample a different one. But because we're imagining that x and x prime are close together, the scatter plot that I get is highly correlated. And as x and x prime get closer and closer together, it should concentrate on the samples being the same.

But then as I move the point x prime further away, now I move x prime over here so x and x prime are kind of far apart, the function values that I get from sampling the functions are now uncorrelated with each other, if it's a Gaussian process with this kind of covariate structure, which is usually what you have.

And basically, it's the choice of what we call the covariance function that actually encodes what we mean by nearby. And for different choices of covariance function, that could mean something else. And just to give you some common examples of covariance functions, one would be to take e to the negative squared distance between the two inputs.

And another, if we have a multivariate case, it could be to take some kind of dot product between the inputs. Basically, the covariance structure is some kind of measure of similarity between the inputs. And we use that to induce what the correlation structure looks like between different inputs. Yeah, go ahead.

AUDIENCE: Does this work with language modeling when you're learning an embedding? Because I guess the words are not continuous, right? You can't take the dot product. But we have an embedding that's learned.

JEREMY BERNSTEIN: Yes. Interestingly, the covariance function-- yeah, if you have discrete inputs-- there are ways to define covariance functions, even on weird spaces. The inputs don't have to be the real line. The inputs to the covariance function can be any inputs that-- so you can extend this construction to weird input spaces.

That was one of the reasons people really like kernel methods as well, because they have that same property. In some sense, a downside of neural nets is it's not obvious how to apply it to language, let's say. OK, we came up with a way to do it. We just embed all the inputs and then learn the embeddings.

But it wasn't obvious that that was going to work before people showed that it worked. So people came up with alternatives. And this is compatible with different input spaces, basically.

AUDIENCE: But I guess, if we were to say that an LLM was a Gaussian process or something, it would mean that whatever the solutions it finds are both a Gaussian process in terms of embeddings, and that it's also figuring out the right embeddings for the solutions of the Gaussian process. Because the embeddings are [INAUDIBLE].

JEREMY BERNSTEIN: So I haven't got to what I precisely mean when I say that we can think of neural networks as Gaussian processes. So let's wait to get there and then discuss this a bit more.

This is the last abstract slide. And then I'm going to start talking about a more concrete example for a neural network. So I hope that will make things a little clearer. But the last thing that I want to talk about is how, if you're doing machine learning with a Gaussian process, how you actually make predictions-- or some people would say, how you do inference, in machine learning speak.

But basically, if you've got training data or you've got inputs and you want to make a prediction at a point that you haven't seen, how do you actually do that? And roughly, the idea is that if you're given inputs x1 to xn and you know the function values f of x1 up to f of xn-- so that's your input data, is the inputs and the function values.

We want to predict f of x-star, where x-star is a new test point, a new test input. That's what we want to predict. You just think about stacking all of the things you know and the thing that you don't know into one big vector.

Because it's a Gaussian process, we know that the joint distribution of this quantity is Gaussian, because that's the definition of a Gaussian process. For any collection of inputs, that includes the n inputs that I was given and the one that I want to predict on, we know that the joint distribution of those things is a multivariate Gaussian just by the definition of the Gaussian process.

And then you can say that, hey, if I have a multivariate Gaussian vector and I know its distribution, one of the standard things that that tells you how to do is, given the first n coordinates, what is the conditional distribution of the last coordinate?

So that's a standard manipulation that you can do with Gaussian vectors is just, what is the conditional distribution of the last coordinate on all the previous ones? So there's just a formula for that. It's just called conditioning with Gaussian vectors or something like that.

And that is precisely the formula that gives you this kind of picture. Because basically, you take this point x-star and you sweep it along this x-axis. And the formula tells you both what the mean of this coordinate is conditioned on these ones. And it also tells you what the standard deviation is.

And interestingly, because Gaussians are so well-behaved, the conditional distribution of the last coordinate conditioned on all the early ones is itself a Gaussian distribution. So if you take a Gaussian vector and you condition on the first n coordinates, the distribution of the last one is Gaussian.

So in particular, there's just a formula for this mean. And there's a formula for this standard deviation. And that's a closed-form expression just involving evaluating the covariance matrix, basically, on the input points.

So I realized I'm not really telling you explicitly what these formulae are, but there are things that you can just look at. And I honestly think if you haven't seen this before, it's maybe easier just to have the intuitive explanation. And then you can just look up what the formulae are if you're interested.

Now I want to tell you what this has to do with neural networks, which was kind of the point of introducing this stuff. And that's a result called the Neural Network Gaussian Process Correspondence, which is something that a lot of people have been really excited about over the years.

So now forget everything we just talked about and just think about a neural network. And we just think the simplest case is arguably just multilayer perceptron. So we've got weight matrices, W1, W2, W3, up to WL.

And what we can do is we can just sample the weight matrices randomly. And it gives me the function. The neural network gives me a function. I write that function down. And then I resample the weight matrices. And I get a new random function.

And I write that function down. And I just keep sampling the weight matrices over and over again. And it's going to give me a distribution over functions. And I can ask, what is that distribution? What properties does it have?

So let's pretend that we did that, and we generated, in this case, 1,000 random functions by sampling 1,000 set of weights for my neural net. So I re-initialize my neural network 1,000 times. And I want to inspect the functions.

So the way I decide to do that is I'm going to pick two inputs, x and x-prime. And in general, x could be a picture of a cat and x-prime could be a picture of a dog. But it's just two inputs. And I fixed them. And then I get my 1,000 functions, f1 up to f1000.

And for those two inputs, I just stack the outputs of the network. So let's just imagine that the output is one dimensional, for simplicity. And I just build a 2 by 1,000 matrix of all the outputs of my random networks. And I'm just going to plot them on a scatter plot and just see what it looks like.

And I actually did this, but I did it two years ago. But I did it. This is a truck that's in the CIFAR-10 dataset. And this was a three-layer MLP. And the width of the network, the width of the hidden layer was 1,000.

And I did it for two pairs of inputs, so input 1 and input 2. So input first of all, and then second, I did it for input 1 and input 3. So input 2 is actually just input 1. It's the same truck, but with a little bit of jitter in the pixels. I just added a little bit of noise to the pixels.

And then input 3, it's the same truck, but I just added a lot more noise to the pixels, just to show the difference. And then for my 1,000 random networks, this is what the network outputs look like for the first pair. And then this is what they look like for the second pair.

So the interesting thing is, it kind of looks like the thing that we were talking about, the Gaussian process thing, where the two inputs, which are kind of similar to each other, seem to have more correlated random outputs, whereas the two inputs that are a bit less similar to each other, the outputs are slightly less correlated.

And then this is displaying it for pairs of inputs. But instead of having two inputs, I could make this same table, but I could put five inputs and then sample 1,000 networks and then plot that five-dimensional scatter plot. And then I could ask, does that thing look Gaussian?

And my claim is it will because of this Neural Network Gaussian Process Correspondence. For a sufficiently wide neural network, these scatter plots actually are Gaussian. And we would actually have to do what Zach was saying.

It's like, how do we actually know these are Gaussian? That's a good question. I don't know. There's tests for Gaussianity that if you wanted to run on this scatter plot and you could do that test. But I don't know how to do that.

But my claim is that really they are Gaussian. And it's because the neural network is a wide network that it has to be the case. So what is the formal claim of the correspondence?

It's something like this, that if we take a neural network and we randomly sample the weights, then so long as the network was very, very, very wide-- if I pick any finite collection of inputs-- basically for any finite collection of inputs, the distribution of outputs is actually a multivariate Gaussian, which is the definition of a Gaussian process.

That was the definition. If you have that for any finite collection of inputs, the distribution of outputs is Gaussian, then that's satisfying the definition of being a Gaussian process. And it also would explain these pictures.

And importantly, what the covariance function is depends on things like what the architecture of the network is and what the nonlinearity is. So it's an interesting result where, given a particular network architecture, you can expand its width and there will be a corresponding covariance kernel or covariance function that is basically a function of what the network architecture is.

And that's what I meant at the beginning when I said that actually the architecture somehow has an opinion built into the choice of architecture about what data points are similar to each other, and which data points are dissimilar to each other.

And one way to extract that information is to take the architecture to be really, really wide. And you extract this kernel function or this covariance function, so this function of the form sigma; x, x-prime. And maybe sigma depends on things like what the depth of the network is, what the nonlinearity is, and so on.

AUDIENCE: So I guess in terms of practicality, because we're making it really, really wide, we're really giving it a good opportunity to overfit. But I guess the idea is because we're taking a random look at overfitting, that's fine? That's OK?

JEREMY BERNSTEIN: Yeah, that's a really interesting question. Let me make some space. Also, I think this lecture is on the shorter side. So the more interesting questions you ask, then the more learning there can be or something. So the question is, if the neural network is really, really wide, then intuitively I would think that it's going to somehow overfit because it has too much capacity for my data.

But the interesting thing about the Gaussian process picture is that, let's say that we have some data and we get this distribution of functions. One interesting function could be the mean function. So remember that not only is there the mean, but these functions are going through the data.

And because it's random and because it's Gaussian, there's some probability of getting a really horrible function, really bizarre. One function could actually do something like, it goes through this data point and then it goes all the way up here, and then it comes all the way back down, and then it goes all the way back up.

And there's some probability of getting that function. But interestingly, it's a very, very small because things can't deviate too much from the mean with very significant probability basically. And even if they do deviate a lot from the mean-- let's say the variance is really high-- well, I can always try to extract what the mean is, because I know that the mean is a nice, smooth function.

So in the context of this correspondence, it's an interesting question to ask-- how can I get the distribution of functions to actually concentrate on the mean? In other words, what do I need to do to the neural network to get this standard deviation to go to 0, relative to the mean, if that makes sense? Is there something I can do?

And I thought about that at one point. And the argument is that if you make the standard deviation that you sample the weights from very, very small, then it actually causes the distribution to collapse onto the mean. And then with very high probability, all the samples are very close to the mean. But I didn't want to dwell. Yeah, it's something to think about.

AUDIENCE: OK.

JEREMY BERNSTEIN: Oh, yeah, there's another answer to that question, which is it's basically like-- I'm not sure if Phillip talked about that in one of his lectures-- but the idea of neural networks being very, very overparameterized, but they can still represent lots of nice, simple functions. So the fact that the network is very, very wide does not imply that you're going to overfit. It just implies that it's possible to overfit. Yeah, go ahead.

AUDIENCE: Why does that require you to have standard weights for the neural network? Also, for example, if you sample 1,000 random network and initially the network is a random Gaussian distribution by sampling it. And [INAUDIBLE] you draw the plot. But the output, you still can't say it follows the Gaussian distribution.

So I'm just wondering, it seems like you can do the same thing in your network for the kernel methods. You can learn the sufficient statistic, like the mean value and sigma value, without a large hyperparameter tuning or without large tuning, how can you learn a sufficient statistic?

JEREMY BERNSTEIN: OK, so let me try to repeat what I think the questions were. So the first one was, in this construction of this correspondence, why does the network need to be infinitely wide? And then I think the second question, if there is this correspondence, why do we actually do deep learning the way that we do it?

Why don't we just extract what the kernel is and then just do Gaussian process regression or do some kernel method? So they're both really good questions. So on the first question, I'm going to talk a little bit about that on the next couple of slides.

It's because the construction relies on a central limit theorem over the width of the network. That's the short answer. So for things to converge to this Gaussian distribution, it's because of a central limit theorem. And that requires something being large. And it turns out it's the width.

The second question, yeah, that's a great question. Because some people really thought that we should do kernel methods. But it just seems to be not the case from the way that we see things right.

It seems that even if you have an equivalent kernel to your neural net, maybe for several different reasons, you would actually rather just train the neural net and just tune the hyperparameters rather than do the kernel method or do the Gaussian process regression.

I think there's probably a few different potential reasons why that's the case. One is, well, first of all, the kernel method, the Gaussian process ends up having hyperparameters as well, because somehow the architecture of the network is going to affect what the kernel is. So it's not like it's really a tuning-free thing.

And then another thing is the kernel method scales much worse with the size of your dataset than neural nets do. I think it's the cost of all the methods is cubic in the size of the training set, whereas somehow neural nets scale linearly with the size of your training data.

I'm not sure if, if you train an LLM, you just train it once on the data. And the computational cost is the cost of a forward pass and a backward pass multiplied by the size of your training set. So it's definitely not a cubic dependence on the number of training points. So that's another thing.

But I don't know. It's a super interesting question why people use neural nets rather than kernel methods. I think it may still be a somewhat open problem. Yes, Zach?

AUDIENCE: I had two questions. So the first is, here we're saying as the width approaches infinity, the neural network is approaching the Gaussian process.

JEREMY BERNSTEIN: Yes.

AUDIENCE: Have people been able to quantify how close a network gets to a Gaussian process if the width is finite?

JEREMY BERNSTEIN: So the question was, if this is a limiting result that only holds at infinite width, if you have a finite-width network, do we know what the deviations are from the limiting result? I think there are papers on this question.

One answer is, even in the classic central limit theorem, there's a result called the Berry-Esseen theorem, which tells you about the deviations from normality of a large sum of independent random variables and so on. So you could just say, I'm going to take the Berry-Esseen theorem and turn the handle and figure it out. I don't quite know if that actually works.

AUDIENCE: What's the name of that theorem?

JEREMY BERNSTEIN: Like, Berry-- I'm probably not getting the name quite right, but Berry, and then I think it's like Esseen.

AUDIENCE: OK.

JEREMY BERNSTEIN: I'm not sure how to pronounce it.

AUDIENCE: And then the other question I had is-- earlier in the lecture, you said that people were really, really excited about this result-- and I'm wondering what the larger effect on the research community has been, how that's influencing how people think about intelligent models?

JEREMY BERNSTEIN: Yes, that's a great question. I think the short answer is that it's had actually very little impact on what people actually do in deep learning, realistically. It may not be the case, but it seems that way. Maybe at the end, I can talk a bit about some of the potential limitations of the result.

But it's like, there could have been a version of deep learning where you're like, ooh, OK, I need to solve this deep learning problem. Let me write down what my architecture is, take the limit, work out what the kernel is, study the properties of this kernel. And, OK, now I'm ready to go.

But that's just not what it is. It's more like, OK, just get an architecture that seems reasonable and we'll just try it. And I think there's a lesson, which is that anything that's too convoluted of a procedure is actually unlikely to be really capturing what's going on.

And one thing is, you just don't need to do that because you can just train the neural net and you just see. But I think there are also limitations of this result that mean it's possibly also potentially not that useful.

So the point of this lecture is not to tell you this is something for your practical toolkit for going out into the wild. It's more like, if you're interested to really get into the technical details of deep learning theory and you want to think about the different ways that people have come up with to think about neural networks, here's one of them. OK. Yeah?

AUDIENCE: How many parameters do we need to optimize in Gaussian process?

JEREMY BERNSTEIN: Well, generally, the kernel, the covariance function could have an arbitrary number of hyperparameters. So if you're doing just straight-up Gaussian process regression, you're going to pick a kernel function. You get to design it and you get to choose how many hyperparameters it has.

AUDIENCE: So what's the most critical kernel function we choose? And what is the correspondence to the neural network architecture?

JEREMY BERNSTEIN: Yeah, I have an example coming up for correspondence to the network architecture. So the question was, is there an example of a typical kernel that people use in Gaussian processes? The first one that most people introduce is just the squared exponential kernel, which is e to the negative x minus x-prime squared.

And then you might introduce a hyperparameter, like dividing by 2 sigma squared, where sigma is a hyperparameter. And it controls the length scale of the correlations that the kernel-- and this is a kernel with one hyperparameter. But then you could say, oh, what if x is a multidimensional input.

Then something you could say is, what if I have one sigma per input coordinate? That would be a way to introduce more hyperparameters. Or you could have a different functional form. You could add something to this kernel. That's the modeling decisions that people incur.

And you can read up papers. And there'll be a table of 10 different kernels. And they'll say, if your problem has this structure, try kernel x; if it has this other structure, try kernel y. It's similar to, oh, use a ConvNet for images and use a transformer for this. And it's just modeling decisions.

AUDIENCE: OK, thanks.

JEREMY BERNSTEIN: OK, I'm going to keep going, because now I think we may run out of time. I'm not sure. So I didn't actually prove it. I just wrote down a sketch. And I think I'm just going to post a reading about how to actually prove this correspondence. But I'm just going to give you something high level about how to prove that this correspondence happens.

So remember, the correspondence is an infinitely wide neural network with random weights induces the Gaussian processes-- the random functions you get are equivalent to a Gaussian process. And the main tool is something called the multivariate central limit theorem, where mainly you just hear about the central limit theorem for a Gaussian random variable with one coordinate.

But it turns out there's a generalization of the central limit theorem to Gaussian vectors. And then proving this result is just applying that theorem a couple of times and doing the right induction over depth of the network. And honestly, it's been a while since I thought about this, but I think there's two steps.

There's two main steps. One is, you pick a single input to the network, and you inspect the network at a particular layer. And you show that the activations for that individual input are iid random variables, which is kind of intuitive that they should be. If you're randomly initializing the network, the different activations of a particular layer on the same input should somehow be iid random variables.

But you have to show that. And then once you've shown that, you then consider that a layer, and you consider batching the activations over the multiple inputs. And you think about each activation coordinate, the batch of examples as being a random vector. And then you apply the multivariate, essentially.

Then you think about the next layer as summing over those random vectors somehow. And you do this multivariate central limit theorem. And anyway, I'm not really explaining it. So the short answer is, there's a central limit theorem. And I'll post a reading if someone wants to go through the details. Yeah, go ahead.

AUDIENCE: The first question is, as the weight goes to infinity, you might be rescaling the weights in some way, right, when you're sampling?

JEREMY BERNSTEIN: Yes, exactly.

AUDIENCE: [INAUDIBLE]. And the second question is, could you talk a little bit more about why they're going to be independent random variables when you have the complex dependencies in the previous layers?

JEREMY BERNSTEIN: OK. So maybe it's not obvious that they are independent. And I think I maybe should not have said that. So I think that's something you've got to show.

And I'm not going to go into it, but it's basically because of the network being so wide at every layer that you have a central limit theorem happening at each layer. And that's going to cause the independence. But I'm not giving you a mechanism, I'm just giving you a hand-wavy justification. I'm going to post the reading.

So the other question was, what precisely is the weight initialization? How does it scale with the width? So I think I have a concrete example of this result in action, so let's just go through it briefly. So I think this is the instantiation of the result for an MLP with a ReLU non-linearity.

And it turns out that all of the calculations, you can solve them exactly for this model. And actually, one of the papers in the reading actually solves this kernel for this architecture. But it's quite an old paper and it may not be that easy to read. So I wonder if I can post something better on Piazza, perhaps.

But basically, we set the nonlinearity to square root 2 times ReLU, and we sample the weights with variance 1 over fan-in. So that's what we would call somehow the standard parameterization in PyTorch, is doing 1 over fan-in. It may actually not be the best way. I actually think it's not the best way to initialize the weights in practice.

But for this statement to hold, that's the initialization you've got to use, so 1 over fan-in variance. And then what you can prove is that the expectation of the network output is 0. This is a network with no nonlinearity on the output. So it makes sense that it means 0.

Because the last layer is a linear layer, the mean should be 0. And then the covariance structure, so the expectation of f of x multiplied with f of x prime takes something called the compositional arccosine kernel. So we take the dot product between the inputs. And then we iteratively apply this function h.

We apply h L minus 1 times, where L minus 1 is the depth. So it's called the compositional arccosine kernel. It's just a particular mathematical function which happens to correspond to this architecture. Somebody proved it. So this is an example of the result.

AUDIENCE: Actually, just a clarification. The whole Gaussian process thing, this is for untrained neural networks, right?

JEREMY BERNSTEIN: Yes.

AUDIENCE: OK.

JEREMY BERNSTEIN: The thing that I told you about, yes. There is another thing, the neural tangent kernel, which is supposed to be a Gaussian process characterization of trained networks. But it also holds under some assumptions.

And people have moved away from thinking too much about that at the moment. But, yeah, there's an analogous result, which is supposed to hold for trained networks. But I didn't talk about that.

OK, I think there's one slide, and then I can take some questions if anyone has more questions. But basically, if you really buy into this picture-- you're like, yes, I believe in the Gaussian process correspondence and I want to see how far I can take this-- you might have questions along the lines of, how does the covariance kernel or the covariance structure depend upon my choice of network architecture?

How does it depend on the way I initialize the weights? And could you actually use these considerations to give you a theoretical way to inform how we pick our architecture and how we regularize the weights during training? Or can this connection actually inform the way that we do deep learning?

So that was the hope for this type of result. And it seems to have just not really happened. Instead, we just do transformer. Or there's a list of architectures, and you just pick one from that list. And so it was a hope of having a formal, analytical way to do design, principle design in deep learning. But it doesn't seem that we actually use it.

So I had one thing, I have a blank slide, there was one thing that I wanted to talk about at the end, which was basically this question of the weight initialization that we talked about. So that result held if we have a layer with a certain fan-in and a certain fan-out, and we use this initialization 1 over fan-in, which is actually the PyTorch standard.

People call this Xavier initialization. But it's not obviously actually the best thing to do, although it is the thing that's built into PyTorch. And I'm just going to give you one quick piece of intuition.

Wait, first of all, why do people believe in this initialization? Why did people come up with it in the first place? And it's basically because it's the initialization that is good at initialization.

And that's the same reason that we use it in the Neural Network Gaussian Process Correspondence, because the correspondence is about how networks behave at random initialization. And basically, this is the way to initialize. If you want to preserve activation magnitudes at initialization, you should initialize this way.

But it ignores a fact of linear algebra that if you have a matrix where the fan-in is much larger than the fan-out-- so the input dimension, the fan-in, is much larger than the fan-out-- then the special fact about this matrix is it has a huge null space, meaning that a lot of the inputs are going to get mapped to 0.

And if you set up things such that at initialization, the activation magnitudes get preserved-- well, at initialization the vectors are hitting the null space, and you're kind of scaling up all the part of the vector which is not hitting the null space. But actually, once you get further into training, actually the inputs to the layer will kind of align with the non-null space. And then this principle is kind of bad because then things are much too large.

Anyway, this is related to something that people call maximal update parameterization or MUP, which is something that if you're trying to train giant networks, people care about this a lot. Anyway, that's the end. So if anyone has questions, I'll stick around. But thank you for listening.