[SQUEAKING]

[RUSTLING]

[CLICKING]

JEREMY BERNSTEIN: Hello. OK, I'm just going to start. So here's a question. And I'm genuinely asking, would you rather to scale the width or the depth of your neural network? Does anyone have any thoughts? Wait, someone said something. I heard someone say depth.

AUDIENCE: Depth.

JEREMY BERNSTEIN: Why?

AUDIENCE: No reason. I know someone said it earlier.

JEREMY BERNSTEIN: Because you've heard "deep learning," so it sounds-- not wider, yeah.

[LAUGHTER]

OK. But it's not really clear. How do you even answer a question like that? I don't know. It's something we got to think about.

So in today's lecture, we're going to ask this question of, what class of functions can a neural network express? And you may have heard this statement that neural networks are universal function approximators. So we're going to think about what that means.

And if a neural network is a universal function approximator and it can approximate any function that we're interested in, then how should we make decisions about the architecture? Does it matter how many layers there are? Maybe a few layers is enough if you make them really wide. Should you have lots of layers?

And ultimately, what do you want to do in practice? And does thinking about neural networks through this lens of universal function approximation actually help you in practice? I'm not going to claim to answer all of these questions, but at least try to provide some beginning of a framework, one framework for thinking about them.

So in machine learning, you can think of it a bit like we have a puzzle. And there are different pieces of the puzzle. We're not going to solve the puzzle until we understand all of the pieces and put them together. And I broke it up here into three pieces.

Let's see, so the first is approximation. This is the question of, does there exist a neural network in my model family that fits the training data? So given my architecture, can it even represent the function that I want, that fits the training data?

The second question is optimization. So let's suppose it does exist. Then that's the question of, can I find it? And third is generalization. So suppose it exists. Suppose I can find it. Then it's like, does it do well on unseen data?

So I really want all of those things to happen if I'm going to solve my machine learning problem. But in this lecture, we're only going to look at the first question about approximation. But just to remind you that there is this puzzle. We really want to solve the whole puzzle and put the pieces together.

So here is a motivating problem. So imagine I have two-dimensional data. It's got two coordinates, x1 and x2. And there's two classes, the red crosses and the green circles.

And I've come up with my, think of this as a one-layer neural net. It's just the w-transpose x plus b. And then I take ReLU of that. If that's my model, can I fit this data? What's your name, sir?

AUDIENCE: Matt.

JEREMY BERNSTEIN: Matt?

AUDIENCE: Yeah.

JEREMY BERNSTEIN: Matt is saying no. Why?

AUDIENCE: Don't you need two linear separation planes?

JEREMY BERNSTEIN: Yeah. Because this thing, there's only one hyperplane defined by w-transpose x. And ReLU does not distort the shape of the hyperplane. It just applies nonlinearity on the output.

This is basically a linear separator. And this data is not linearly separable. So this is supposed to be recap from lecture one. So hopefully, it's reasonable. But we cannot linearly separate this. But what if I had a two-layer neural net?

Do you think I could maybe do it with two layers? Potentially? Yeah. OK. So I think we can do it with two layers, but you may need to think about it. But this is just supposed to be a refresher. But it's saying we've got a kind of function that we're trying to fit, and we have a model family.

And we're asking, does there exist the function in that model family that can fit that data? So it's a kind of warm-up for thinking about this idea of function approximation. I have a second motivating problem. Does anyone recognize this function? You recognize it? What is it?

AUDIENCE: Isn't it Weierstrass' function?

JEREMY BERNSTEIN: Yeah. It's Weierstrass' function, which is everywhere continuous but nowhere differentiable. It's a classic example of a pathological function. And the question I wanted to ask is, do you think you could fit this function with a neural network? Does anyone have a thought? Matt?

AUDIENCE: Wouldn't you have to go to infinity? It would be like asymptotically, I guess?

JEREMY BERNSTEIN: It seems like you would need some kind of asymptotic thing. Yeah, honestly, I don't know what the answer is to this. I just thought it was an interesting function. And it's interesting to think, can I fit it with a neural net? I guess I suppose the answer is, maybe.

And I thought a good final project, if someone wants an idea, is to take all these pathological examples that Weierstrass was thinking about and see if you can fit them with a neural net. But anyway, this is just more motivation, just something to think about.

OK, so this is a fractal function. It's defined recursively. It's self-similar. If you zoom into any piece of the function, it resembles the whole function. So we want to now formalize this approximation problem that we've been talking about.

So given a family of curves G-- we think of these as the curves that we want to be able to approximate ideally. And we can choose what we want G to be. We could choose it to rule out Weierstrass' function. We could ask that G is differentiable.

Or we're free to pick whatever family of curves we're interested in. That's up to us. And then we think of a family of neural networks F. This is like, I'm going to say, I care about a five-layer MLP, a five-layer multilayer perceptron. That could be my family.

So in other words, a neural architecture could specify a family of functions. And then I want to say, if I pick any curve in my space of functions or set of functions that I'm interested in, does there exist a neural network in my family of neural networks that can fit that function to some small error?

And we can come up with different notions of error. And OK, here, epsilon, you should think of epsilon as a small number. So we choose epsilon, and we ask, can we approximate any function in the family of curves G to less than some small error? And we have to ask, what do we mean by the error?

So we could pick whatever metric we want. So one example, we could call the L-infinity error, which would be the max over inputs of f of x minus g of x, and then take the absolute value. So this, think of it like L-infinity norm, but for functions. So it's the max discrepancy between the two functions anywhere on the input space.

But another equally valid error measure, we could call this the L infinity measure. Another measure would be the L1 measure, which would be the integral over the input space of the absolute value of f of x minus g of x. So that would be another valid way to measure discrepancy, which is, I integrate over the function and just sum up all the differences along the whole axis.

So in this lecture, just for the purposes of having something nice to present in the lecture, we're going to pick a particular family of curves G, that we're interested in whether neural nets can approximate this family. And these are the Lipschitz continuous functions.

So has anyone-- well, I think it's nice to teach you about Lipschitz functions regardless of neural net approximation. So that's another nice reason to tell you about this. So let's just think what a Lipschitz continuous function is. It's a particular notion of continuity.

So what we're going to do is we're going to say that a function g from real numbers to real numbers is L-Lipschitz. So L is a number. It's like, 10 or 5. But it just measures how Lipschitz the function is.

We'll say it's L-Lipschitz if for any input point x, the absolute value of g of x plus delta x minus g of x-- so think, if I change the inputs by a delta x, I'm going to change the function. And the amount that the function changes is bounded by the Lipschitz constant times the size of delta x.

So this we call Lipschitz continuity of g. And there's an intuitive way to think about what this is saying. Imagine taking delta x, take the limit that delta x becomes really small. Can anyone recognize what this looks a little bit like, if we take the limit that delta x becomes small?

AUDIENCE: Is it that the slope is always less than some constant L?

JEREMY BERNSTEIN: Exactly. Yes, exactly. So you should recognize that if I take the limit that delta x goes to 0, it's a bit like saying that the derivative of g is bounded by L. You should see a definition of the derivative hiding in here. So Lipschitz continuity is a kind of generalization of the notion of having a bounded derivative. Well, it may be equivalent actually, but you need to think about that.

And just to draw a picture, let's suppose that we set x to 0. So we think about this x as being 0. And then we ask how large can g of delta x be. And what this definition says, if the function is L-Lipschitz, we think about drawing the line like y equals Lx.

And similarly, we draw the other line y equals minus Lx. And this definition implies that whatever function we have-- oops, it's not supposed to be that squiggly. It should be reasonably not too-- anyway. The function had better lie within those bounds.

So the implication of the definition, if the function goes through the origin, it has to lie within those bounds. But the same argument applies at any point. So it places a cone kind of shape maybe, or a kind of bow tie. And the function has to belong to that bow tie at any point along the function, if that makes sense. You imagine translating the bow tie along.

So that's Lipschitz continuous. And now I thought, well, another great thing to introduce you to is how to generalize that notion to functions with multiple inputs. So now we're going to think about generalizing this Lipschitz notion for g going from Rd to R. And basically the inputs are now going to be in Rd, and the delta x's are going to be in Rd.

And you see we don't really need to change anything here. The output is still one dimensional, so that still makes sense. But this doesn't really make sense, because you can't take the absolute value of a vector. Or maybe you can, but that's not what we want.

What we're going to do is we're going to change the absolute value to the norm of the vector delta x. And I'm going to pick a particular norm that I really like called the RMS norm. So now, do you see that this is now a valid definition, as long as I define what I mean by the RMS norm?

So the idea is that because the x is a vectors, we need to measure the size of the vector on the right-hand side. So I'm going to define the RMS norm of a vector x to be the square root of 1 over d, which is the dimension of the vector, times the sum of the coordinates xi squared, which we can also think of as just being 1 over square root d times the Euclidean norm of x.

So I just want to introduce you to the idea that if you have an object like a vector, there's many different ways to talk about how large it is. The Euclidean norm is one example, but the Euclidean norm is kind of a dimensional object. You can think about the RMS norm as a kind of non-dimensional analog of the Euclidean norm.

Because if all the entries of the vector are 1, then the RMS norm is 1, whereas the Euclidean norm would be like, square root d. But anyway, the point of this slide is just to generalize Lipschitzness to functions with multiple inputs, and to point out that there's many different possible ways to measure the size of something.

OK, but why were we even talking about this? The point is that in this lecture, in the first part of this lecture, we are going to prove something akin to a universal function approximation theorem. And the class of functions g that we're interested in approximating is going to be the class of L-Lipschitz functions that map from the hypercube.

The domain or the input space is the hypercube to the real numbers. So this is the hypercube. And these are the reals. And we're only considering L-Lipschitz functions, so functions with bounded derivatives.

And what we're going to say is that if we pick an error tolerance that we're interested in, then there exists a three-layer ReLU network with a certain number of units such that this integral notion of error-- summing up all the little pieces and adding up all the absolute values of the difference between our neural net and the function g-- that integral is less than apparently 2 epsilon.

Does anyone have any questions about this? We could dwell on it for a moment. This is our goal basically for the next 20 minutes. Hopefully, we're just going to try to prove this statement. Why should we be interested in such a statement? That's another question. Yeah, go ahead.

AUDIENCE: The 0, 1, the mapping from 0 to 1 definition, is that taking only values from 0 and 1? Is that what that means?

JEREMY BERNSTEIN: The interval 0, 1, with square brackets refers to the interval of the real line just between 0 and 1. So that's just picking out an interval on the real line. And then raising it to the power d means that in all the dimensions of our d-dimensional space, we pick out that interval.

So it's like picking out a cube in three dimensions or a hypercube. It's just supposed to be an abstract way of writing down the hypercube in d-dimensional space, where all the axes are between 0 and 1.

AUDIENCE: What is N here?

JEREMY BERNSTEIN: N? Capital N is my symbol for the number of units. By units, we mean the number of neurons in the ReLU MLP. So N is the number of neurons in the three-layer network.

AUDIENCE: OK.

AUDIENCE: I have a couple of questions. So first, are those neurons per layer or are they the total number in the whole network?

JEREMY BERNSTEIN: It's the total. But honestly, because it's three layers, for the purpose of understanding the lecture, you can just pretend that the number 3 is the number 1. And that kind of question doesn't really matter so much. But I really mean actually the sum total of the neurons.

AUDIENCE: So a second question is, when you say three-layer ReLU network, what precisely do you mean by that? So are there three value functions in it? For example--

JEREMY BERNSTEIN: I precisely mean the thing with three weight matrices and two value functions that follow the first weight matrix and the second weight matrix. And then the third weight matrix doesn't have a ReLU.

AUDIENCE: OK. And then the final question I had is, this theorem seems very specific. I was just wondering if you could speak a little bit as to what you brought up-- why do we care about this?

JEREMY BERNSTEIN: Yes.

AUDIENCE: [INAUDIBLE]

JEREMY BERNSTEIN: So that's a great question. So the reason is because the way to prove this theorem is not super involved. So the point is to show you an example of such a theorem. And you can see the whole proof and how it works. And then I'll point you to some other references where they do different calculations or different things.

It's just to show you an example. And then we're really going to think about, is this relevant? I'm not claiming that this is an important result. I'm just saying it's something you can prove. And then we'll think a bit about, is this an important result or not? Yeah?

AUDIENCE: Does this theorem apply to networks that are more than one layer? In our notes from the first lecture, there was a section about universal approximation theorem for a single value layer, where you have a given amount of value units.

And in that single layer, you could approximate a function to arbitrary precision. So does that apply? Does this theorem apply to layers that are more than one?

JEREMY BERNSTEIN: This applies to a three-layer ReLU network. So this is not the most general thing that you can prove. The result that you're talking about is a different result. This is just another result where there's a proof, which we can put into some slides and show to you.

It's just to get you thinking about how such a result could look. But there are other results. There are probably more interesting versions of this result. This is just a result, but it's about three-layer network.

AUDIENCE: So for this, is it possible to shift the interval or scale the interval?

JEREMY BERNSTEIN: Yeah. Anytime you see the hypercube, you should think, I can probably just rescale that. And I'm going to change some scaling numbers. I'd probably change maybe what the Lipschitz constant is. Yeah, you should be able to rescale it to be any hypercube that has arbitrary sizes in different dimensions, basically, or hypercuboid or something like that. Yes.

AUDIENCE: Why do we want to constrain everything to cube? Does this mean that everything should be finite?

JEREMY BERNSTEIN: Yeah, that's an interesting question. Is that a toy thing about this theorem? But actually, if you think about in real deep learning, you usually normalize your inputs to be coordinate-wise, about 1 in magnitude.

You could actually train a neural network on an interesting problem and actually project all the inputs to live within a hypercube, or maybe-- a hypercube. Yeah, yeah. So that part is actually a realistic assumption that my data lives in a hypercube, for many problems, not for all problems. Yeah, last question.

AUDIENCE: So the 1 over epsilon to the d seems pretty important. Do you know if you relax the L-Lipschitz that will take a different definition of continuous, do you still remain with that as the best you can do? Or can you do better than that power law?

JEREMY BERNSTEIN: Yeah, I imagine with more knowledge about the problem structure, you can do much better. I'm not trying to claim that this is an interesting result. And we'll talk about the limitations.

And I think you would hope to do-- yeah, just look. It's saying that you need N-- the number of neurons you need, you take the Lipschitz constant, and you raise it to the power of the dimension, which is really, that could potentially be massive.

And also the smaller the error of tolerance you want, the bigger the number of neurons you need in a way that depends exponentially on the dimension. So that's really bad dimension dependence. And we don't, in practice, really want to do that ever.

Yeah, it's great to point out that the result itself is not even the best possible result we might hope for. Whether we can do better probably depends on structure in the problem. And I don't know even if under these conditions, maybe there's something better, but I'm not sure.

OK, let's just plow ahead now for a little bit. So the strategy that we're going to adopt for proving this result is, first of all, we're going to pretend that our inputs are one dimensional. We're just going to treat the case of one-dimensional inputs, one-dimensional outputs.

And we're going to forget about ReLU networks for a little bit. And we're going to think about approximating our function just with rectangles that we build up. And you can see that you can approximate a function by taking these rectangles and putting them like this.

And you can see that as you make the width of each rectangle smaller, you get a better and better approximation to your function. And that's something we can quantify quite easily. The second step is then we're going to generalize that rectangle construction to multidimensional inputs, still ignoring ReLU networks.

And then the third step is we're going to show that with a two-layer ReLU network, you can approximate a hyperrectangle of that kind. So then you can just use the third layer of the network to linearly combine all of your rectangles and you'll be able to approximate your function.

So this is the strategy. First approximate the function with rectangles, then approximate it with hyperrectangles. Then show that a two-layer ReLU network can approximate a hyperrectangle. OK, let's just plow on, in the interest of time.

So the first step is to construct these rectangular strips. Each strip is centered on a discrete grid point. And you can think that this is like building a function-- f of x is the sum over alpha i times, let's call this the i-th indicator. What you should think is that alpha i is measuring the height of the i-th rectangle alpha i.

And indicator i is the function, which is 1, if the input is in this interval, and 0 otherwise. And then if I sum up a bunch of these indicators of this form, that corresponds to this approximation to the function. Is that clear? OK.

So the next step is basically what we're going to ask is, if we approximate a function in this way-- let's say we make the left edge of the rectangle actually touch the function. We need to ask, how big is this gap? How big can it be?

And the observation is-- OK, does anyone have an idea of how to place a constraint on how large that difference can be? Yeah?

AUDIENCE: We use the Lipschitz condition. It would just be the area of the triangle.

JEREMY BERNSTEIN: Exactly. So the observation is that we're assuming that the function g that we're trying to approximate is L-Lipschitz. Like we said, that places a bound on how large its derivative can be. And that gives us a constraint on how big this triangle can be. Or the triangle could go under, but it tells us how big this thing can be.

And so we're going to ask, given N strips, where N, we can ramp it up or we can make N as big as we want. But given that we have N of them, what is the approximation error? And the claim is that the approximation error is bounded by the Lipschitz constant divided by 2 times the number of strips.

And so importantly, if the Lipschitz constant gets bigger, then if I keep the number of strips the same, the error would get bigger. But if I make the number of rectangles larger, the error would go down.

So to prove this, can you see that basically, we just need to do the geometry of what this triangle can look like and then sum it up over all the strips? So let's just do that quickly. So if we have N strips, each rectangle is width 1 over N.

And the height of the triangle, if you just think through the definition of Lipschitzness, you just multiply the width of the strip by L. And that will give you the maximum possible height. And then, of course, then the triangle has area-- What's the area of this triangle? Can someone tell me?

[LAUGHTER]

Oh, wait I remember. It's that 1/2 L over N squared, right? The 1/2 base times height. And this means that the total error-- remember that we're dealing with the L1 error of the function approximation-- is just going to be like N times the error of each rectangle.

And that corresponds to actually the area of the triangle, the maximum possible area of the triangle, which is 1/2 L over N squared. So this is just 1/2 L over N. OK, now we just need to flip things around. That's the largest possible thing the error could be.

But what we wanted is a statement of the form-- if we have this many rectangles, then the error cannot exceed epsilon. So we need to flip the statement around. So that corresponds, the way you do it mentally is just you say, epsilon is 1/2 L over N.

So then to achieve error epsilon, we just rearrange this equation. We get N needs to be greater than or equal to 1/2 L over epsilon. And if I don't care about this factor of 1/2, I'm free to just also erase that factor, because it doesn't really matter.

The form of this statement, is if I set N greater than L over epsilon, then I will have an error epsilon less than 1/2 L over N. So I'm just rearranging the equation and being careful about the inequalities. So this is step one. We know how many rectangles we need to get a certain error in the one-dimensional case with one-dimensional inputs.

Let's think about how things change if we have multidimensional inputs. And the thing that changes is we now have this hyperrectangle, just think 3D. Whenever you need to think about high dimensions, you just think about things in three dimensions.

What I'm doing, I'm just repeating the calculation, the total error, and then I'm breaking it up into pieces. So this piece is just the number of hyperrectangles. This piece is the error per hyperrectangle. No, we think about the hyperrectangle as having a little cap on top, where the hyperrectangle differs from the surface G.

And we know the surface area of the top of the hyperrectangle, because it's just the width of the hyperrectangle raised to the number of input dimensions. So we know the surface area of the top of the rectangle. That's easy. And then we just use the Lipschitz constant to measure the maximum possible height of the hyperrectangle.

So it's just repeating exactly the same thing that we did, except instead of the error being a triangle, it's some kind of generalization of a triangle. So I'll probably just leave people to think that through by looking at the slides, if you want to think more about that. But this is the height. Let's call it the error cap.

Is this right? Yeah, it is right. And then if you think about, what's the surface area of the top of the cap? Well, if there's N hyperrectangles, if I sum up all the surface areas of all of them, I should get 1.

Yeah, so then the surface area of each one should better be 1 over N. So we'll just call this the area. If anyone wants to think through this more carefully, please do so. And the claim is that we've just done the same calculation.

The only remaining thing is to rearrange it in the way that we just did. And this would say that we should need L over epsilon to the d. So I do the same trick of setting epsilon equal to this thing, and then just rearranging.

Now, let's just zoom out and just try to assess where we're up to. So what we're imagining is that we now have a function g with multiple inputs and one output, which we can think of as a surface. That would be the case where there's two inputs and one output. It's like a surface. And then in higher dimensions, it's a generalization of that.

And the claim is that if we want to approximate this with N hyperrectangles and we want to get a certain notion of the integral of the error, that the number that we need to get an error epsilon is L over epsilon to the d. And this should remind you a lot of the thing which appeared in the theorem statement.

And the reason for that is that the next step is to show that we can approximate the rectangle with a ReLU, with a two-layer ReLU network. So the number of small two-layer ReLU networks we're going to need is L over epsilon to the d. And then we just need to count how many neurons are in each of those ReLU networks.

So I just decided to just zoom over this. Because I want to show you actually how to construct this thing. But basically the claim is that there's a one-layer-- is this one? OK, we would call this a two-layer ReLU network with a parameter c, a kind of weight c.

And if we take c to infinity-- which seems like a bit of a trick, but OK, that's what we're going to do-- the claim is that that network converges to this function, which is exactly this single rectangle in one dimension. So I just want to take a bit of a risk by switching off and hoping that I'm able to reshare my iPad. I'll just try to build this thing for you.

So this is a ReLU. By the way, this kind of manipulation is quite helpful on some of the homework problems, probably. This, I really recommend. I'm not being paid by this graphing software, but it's really great. And what I want to show you is, so here I've got ReLU of x.

And then I'm just going to subtract ReLU of x minus 1. And you see it flattens it out. And then say I want to curve it down, so I subtract another ReLU. And then I want to flatten off that little flat bit that's not flat. So I'm just going to add a ReLU. And each time, I'm translating them one along.

But now I'm like, hey, that doesn't look like a rectangle. The slopes are too non-slope-- I want them to be slopier or something. OK, I'm going to insert a constant c, which is initially 1 so it's not changing anything. And I'm just going to use this constant to increase the slope of all the curves.

And let's put it between 1 and 10. And then I'm just going to increase it. And I'm like, oh, it's making the slopes slopier, but it's also squeezing them together. And so I can solve that problem just by translating them back, I hope. That doesn't look like what I wanted.

Let's see. Oh, now it looks like what I want. So I just translated them back. And then I'm just going to let c get a little bigger, maybe 1,000, and make it a bit bigger. And you see, OK, I did it. That's just combining ReLUs. That's technically a two-layer neural net. And I could approximate a square function.

So that's what this slide is doing, which I now hope I can bring back. So that is the demo of approximating. And you can see that using that graphing software, you can just play around with things that you can figure it out. Go ahead.

AUDIENCE: What's the relationship between a two-layer ReLU network and a Riemann sum?

JEREMY BERNSTEIN: Yeah, so we're going to use one copy of this two-layer network to approximate each rectangle, to approximate one of the rectangles in that Riemann sum kind of thing. And then we're going to weight each. And you have to insert factors to translate it to the position that you want.

So you translate it, and then you multiply it by a little scalar to get it to be the size that you want. And that scalar would correspond to the third layer in the network, to set the size of the rectangle, how tall it is or what its height is. Yeah?

AUDIENCE: Why do we restrict ourselves to L-Lipschitz functions? Wouldn't that be for any kind of function?

JEREMY BERNSTEIN: Yeah, it's to get a sense of the error when things are finite. Because we're not interested in actually taking the limit that the number of strips goes to infinity. We want to know for a finite number of strips. So at least that's how we're using the Lipschitzness.

But, yeah, I think it's necessary. I'm not 100% sure. But we can maybe talk about that after the lecture. Let me just keep going, because I'm a bit worried about time. This is just half the lecture. And I guess we're nearly halfway through the lecture, so maybe we're doing OK.

So that was a one-dimensional rectangle. But remember that we need to be able to approximate a rectangle with a hyperrectangle. In two dimensions, it's a kind of cube. And in three dimensions, it's a hypercube kind of thing.

OK. But there's another trick. There seems like there's a lot of tricks going on here. But another trick is, we basically take a hyperrectangle that's aligned with one axis, and we take a hyperrectangle that's aligned with the other axis.

And we realize that only in the place where they're both kind of on, if that makes sense, where they cross each other, the height of the thing, if we just add them, is 2. And then the height of the thing where only one of them on is 1. So do you see that's this picture?

We add them and we get this plus-shaped surface, where in the very middle, it's height 2 and in the other places, it's either height 1 or height 0. And so the observation is that we can then add these rectangles in one dimension, subtract 1, in this case, or in general d minus 1 and bring the surface down.

And then the only place which is kind of coming above the axis is the place where they're all intersecting. And then we're just going to threshold. We're just going to apply ReLU on top of that. And it's just going to pick out the place where they're all on.

Does that make sense? So they'll only all be on where they all intersect. And then we just shift the whole thing down and threshold, just to slice off the top. And it's going to give us that little cube by just slicing off. And we can do all of that just by adding and doing a ReLU. So that's the logic.

And now we're ready to assemble all the pieces. So the top thing is the two-layer ReLU network that can approximate a rectangle. We can then sum these things over dimensions, subtract d minus 1 like we just said, and then take a ReLU. And this will now give us the hyperrectangle that we always wanted with this constant c.

c is the thing that we're going to take to infinity to make the slopes slopier. And then finally, we're just going to take linear combination of these things with constants alpha i, which are going to set the height of each hyperrectangle. That corresponds to adding one more layer on top of our network.

So you can see that in this construction, there's a ReLU here and there's a ReLU here. So it really is a three-layer network, by the definition that we gave. And then the final step is just to let this constant c grow really large so that we really do approximate the hyperrectangles. And we can approximate our arbitrary surface.

Let's just recap where we got to. We were going to try to prove this theorem. And the general idea was we're going to approximate these-- you could call them bumps or these hyperrectangle functions-- we're going to use ReLU to approximate those things. And then the final step is going to be to take a linear combination.

So I'm going to dwell here a little longer. The intention of this slide was to think about some of the limitations of what we just did. I feel like some of them you already raised like throughout. But does anyone want to discuss anything or point out? Oh, yeah, go ahead.

AUDIENCE: So the two different layers of ReLUs do very different functions over that?

JEREMY BERNSTEIN: Yeah, one of them is about approximating the rectangle to get the slopes of the sides. And the other ReLU is more about slicing off the top of the hyperrectangle.

AUDIENCE: OK, great.

JEREMY BERNSTEIN: Yeah?

AUDIENCE: Has there been any research in determining if ReLU networks actually do something?

JEREMY BERNSTEIN: Well, yeah. Let's talk about that on the next slide, because I'm pretty doubtful. It seems like quite a toy construction-- not a toy, but it has a very deliberate intention, which is proving this approximation thing. And a lot of the steps seem a bit kind of fishy, in terms of would this be something that a neural network actually does, like letting the constant c tend to infinity?

Usually you don't want the weights to blow up. That would be a sign that something's going wrong in your neural network. So I wanted to list here a couple of these. "c tends to infinity" seems a bit questionable. Yeah?

AUDIENCE: [INAUDIBLE] going through the derivation [INAUDIBLE]. So a few slides back, you set f of x equal to the sum of all these rectangles?

JEREMY BERNSTEIN: Yes.

AUDIENCE: But also from my understanding, at this close approximate g. And so I was confused how the sum of the rectangles is associated with this [INAUDIBLE]. Does that make sense?

JEREMY BERNSTEIN: Yes. It's because of the use of these. We're thinking about it symbolically, as being a weighted sum of indicator functions. So you think of the rectangle as being a function, which is 1 if the input lies in this interval, and 0 otherwise.

AUDIENCE: I see. So I thought it said i and not the width of an indicator. That clarifies it.

JEREMY BERNSTEIN: Yeah, sorry, I didn't spell it out. There's an annotated version of the slides on the website where it's a bit more spelled out. But exactly, it's not an interval there. It's an indicator.

AUDIENCE: And so if it's an indicator, then we're summing the height at each point in the rectangle?

JEREMY BERNSTEIN: Yeah. The way to think about it is, for a given input, let's say this is our input, only this indicator is triggered. And then it gets scaled by its height. So there's a sum of terms, but only one of them will ever be active.

AUDIENCE: I see. That makes sense.

JEREMY BERNSTEIN: OK, one last question, please.

AUDIENCE: How did you get the 4D [INAUDIBLE]?

JEREMY BERNSTEIN: Yeah, the 4D is about counting the number of neurons in this construction. And the point is that in the top purple bit, the rectangle, there's four neurons. Then, in the hyperrectangle, I'm summing over d of those terms, so it gets 4 multiplied by d.

And then so each hyper rectangle needs 4d neurons. And then I have the final term, which is the number of hyperrectangles I need. So that's why it's 4d. Yeah, I hope it's right because I just tried to work out the constant yesterday. But let's see.

OK, let's just go ahead. So I wanted to think of it. Is this really realistic of what would actually happen in training? One of the questions is, if we're just trying to say that a ReLU thing is going to approximate a rectangle, then why wouldn't we just use a rectangle as our basis function to begin with?

Because there's something kind of circuitous about not just doing that. So let's just pretend that we've created a basis of these rectangles and that someone gave us some data, some x's and y's, and we're going to fit it. And just think about how the training would go.

And what I'm claiming is basically, every time we see this data point, this rectangle would get drawn down towards it. And every time we see this one, this one would. And every time we see this one, this one would. And every time we see this one, we get this one.

But there's all these other ones which are never actually going to move. And so it's really kind of unnatural. Visually, the obvious way to approximate this data is just to draw a line through it, not to take rectangles at each point and move them up to the height of the-- So there's something unnatural about it.

But this is the construction that we use to prove the theorem. In particular, the thing I'm pointing out here is that it seems like if you actually fit data this way, the training performance is going to be really good, but the generalization performance is going to be really bad. Because you're just fitting the data on tiny little strips, which doesn't seem amazing.

OK, just to reassess where we're up to, the goal was just to introduce you to one particular function approximation result, of which there are many. There are many papers on this topic. And there are some classic papers on this topic. So one is called Barron's theorem, which uses something about the Fourier representation of the function that you're trying to fit.

And there's a classic result. So our construction there was using three layers. There's a classic result which uses two layers to do universal function approximation, and actually a more powerful result, from my understanding. And it uses something called the Stone-Weierstrass theorem, which I thought was interesting.

Because Weierstrass was this mathematician who, I think he was alive in the 1800s, but I'm not 100% sure. But he came up with that fractal curve and was thinking about these analysis questions. But he was actually, seemingly thinking about approximation as well. And he had a theorem, which was like, you can use polynomials to approximate any continuous function or something like this.

And it's just kind of interesting that this branch of math is related to a very classical branch of math. But that's all I have to say. So on this slide, I want to ask-- is the thing that we just spent 45 minutes talking about, is it actually important? And here I'm just abbreviating UFA to be universal function approximation. Uh-huh?

AUDIENCE: Where does this lie-- I guess the universal function approximation theorem-- lie in accordance with using Taylor's approximation-- or Taylor's expansion of a function and fitting a neural network using polynomials for [INAUDIBLE] function?

JEREMY BERNSTEIN: Well, that would be another way of just fitting the Taylor expansion up to some degree, which is a bit like that Weierstrass thing of using polynomials to fit a continuous function. It's like a valid mathematical way to approximate a function, using a Taylor series.

But it's just not what we're doing in deep learning. In deep learning, we take a neural net. It's just different. Thinking about that question more, I think is great. I don't have any really great comments. Yeah, putting all of these different ways of thinking about things and trying to reconcile them is like a really productive thing to try to do.

So the point of here was to say, is universal function approximation sufficient? Is it like, OK, that was lecture 3-- now for the rest of the class, we'll just do applications now, because we understand the theory of deep learning? Probably not.

So I wanted to give examples of other things which are universal function approximations. So other examples would be polynomials. Can anyone think of another example of something which seems like it's probably also a universal function approximator?

AUDIENCE: Random search. [INAUDIBLE]

JEREMY BERNSTEIN: Random? Yeah, how would you parameterize the space of functions?

AUDIENCE: So in the previous example when we had the hypercube, we can just again segment it into N locations. And then when we train, we just get stuck at whatever point we got right.

JEREMY BERNSTEIN: Yeah. So here we're not really doing training. We just want to think about what is the space of functions. So the space of functions you talked about there is linear combinations of these hyperrectangle bumps, which that's valid. It's a bit wordy, so I'm not going to write it down, but that's valid.

The other one I wanted to point out is something like the space of Python programs. So programming languages are, in a sense, a universal function approximator. Does that mean that we've solved machine learning because program-- it's like many, many things are universal function approximators.

It's just a piece of the puzzle. That's the thing I'm trying to point out. Just because you've installed Python, you can't start doing machine learning. You need something else. And is it necessary for machine learning to work? So if your machine learning model is not a universal function approximator, should you be worried?

Someone's saying no. Yeah, I sort of agree, but I'm honestly not sure. Yeah, I think it's just an interesting question to ask. Do we need it to be? My guess is not. My guess is probably we don't need to have a universal function approximator to do machine learning. But I think it's a little unclear how to answer that.

So now I want to move on to the second half of the lecture, which is going to be thinking more about that question of width versus depth. So again, would you rather scale the width of your model or scale the depth?

And again, we've just tried to prove in some sense that with a very small depth network, like depth 3, we can fit any function. So that's coming back to this question of, if three layers is enough, why would we ever want to have a deep network? Does anyone have a thought? What do you think? Yeah?

AUDIENCE: I'm guessing because in the terms of nailing the network and stuff, you can approximate, I'm guessing, some sort of arbitrary nonlinearity, because you have pointwise nonlinearities in each of the neurons, you have function compositions. So those would help you to do a complex mapping. But width would be going back to the universal approximation theorem.

JEREMY BERNSTEIN: Right. That's great. Let me just summarize. I forgot. I'm always supposed to summarize questions and comments, but I forgot about that. But basically, if we layer lots of layers, then we have a kind of compound effect. We have more nonlinearities. It seems like maybe that can give us a richer space of functions through compositionality. Yeah, that's a great point. Sorry, what was your--

AUDIENCE: Maybe on surface what I think we can have is we can put on lines and dots. The width might be faster or take less memory. But if you don't, then that might be good.

JEREMY BERNSTEIN: This is a great point that, if you think about building a machine learning system, width can be parallelized. It makes better use of parallel hardware. So from a systems perspective, if you can get away with a shallow network that's very wide, you would really love that, because you can parallelize it.

And that's illustrative that there's different constraints. There's the, can I represent complex and interesting functions? And then there's the computational constraint of, how efficient is this to run? There's other constraints of that sort.

So on this slide, I wanted to just think, what are the advantages of width? So one is parallelism. One is a three-layer neural net is a universal function approximator, or maybe even two layers is enough. Can anyone think of any other advantages of width?

AUDIENCE: You get the feature representations that are nicer and more easily interpretable.

JEREMY BERNSTEIN: Mm-hmm, easier to interpret. I'm trying to remember if there were any others that I thought of. Does anyone else have any?

AUDIENCE: It appears [INAUDIBLE] gradients.

JEREMY BERNSTEIN: Yeah, easier to train, that's a good point. Easier to optimize. Yeah, because whenever you do compound anything, it is very liable to get out of control. So making your network too deep without being careful about how you do that can actually just break the training.

So if you just take a vanilla multilayer perceptron and make it 50 layers deep and just try to train it, you'll see it's very difficult to get it to train. OK. I can't think of any others, unless anyone else had. So the point of this slide was to say, well, OK, obviously wide is better than deep, right? That's obvious now.

But then the next slide is supposed to say, is it? Because now in the last part of the lecture or almost the last part of the lecture, we're going to think about something called depth separation results, which is another potentially quite stylized theoretical result, but it is quite interesting.

So remember that the universal function approximation results suggested that we need to make our network exponentially wide in order to be able to approximate everything that we want to. And that seems bad. So the point of a depth separation result is to say that there's actually a deep network that can represent a particular function quite efficiently.

And if you tried to represent the same function with a three-layer network or a shallower network, you would need exponentially more neurons to do that. So the shape of a depth separation result, I'm just trying to break it up into stages, is first of all, we pick a property of a function.

And the example we're going to think of is the number of linear regions. We think of the function as being piecewise linear. And how many linear pieces are there? And then we construct a deep network that has a lot of that property, but is not a particularly big network in some sense. It doesn't have a lot of neurons.

And that is a constructive thing. We say, here is a neural network which has a lot of this property. And then we say, now we're going to prove that it's actually impossible for a shallow network to have that same property unless it has exponentially many neurons or a huge number of neurons.

So that's the shape of this result. And again, there's a literature which just proves different varieties of this kind of result. And that's called depth separation results. We're just going to show you one particular example.

So recall that a piecewise linear function is a function where if you break it up into segments, each segment is linear. And then they kind of stitch together.

And we'll think about defining the number of kinks in such a function to mean-- a kink is how many of these discontinuities are there in the derivative, how many places does the derivative suddenly change. And so for this function, you can see there's four kinks. And that's the property that we're going to pick.

And we're going to show that there's a deep network with not very many neurons that has a lot of kinks. And if you want to have that many kinks in a wide network, you need exponentially many neurons. That's the strategy of what we're going to try to do. But the first claim is that ReLU networks are piecewise linear.

So if you have a ReLU network with one input and one output, it will look something like this green curve. Can anyone tell me why that's the case? Or does anyone think it is the case? Or does anyone think it's not the case? Yeah?

AUDIENCE: It's because the ReLU is essentially a linear function, but with a threshold. So when you add them, it creates some kind of shape. And the transformation that you did before is linear, so it's only going to distort it but in a linear way.

JEREMY BERNSTEIN: Great. Yeah, I think with this kind of statement, with many sorts of statements-- yeah, exactly. But with many sorts of statements, we want to break them up into-- I'm not sure what the right word is. But we want to be able to say, if f and g are both piecewise linear, then that implies that f composed with g is piecewise linear.

If f and g are both piecewise linear, that implies that f plus g is piecewise linear. If f is piecewise linear, then alpha f is also piecewise linear. So you break up the construction of the function into a series of combinations, and you prove that the property is preserved under those combinations.

And then that and then basically you realize that, OK, a ReLU neural network, the ReLU function is piecewise linear. And then it's just adding and multiplying things. So you're always going to get something that's piecewise linear.

So based on that, let's now construct a depth separation based on counting the number of kinks. So I'll first give you the intuition and then write something a bit more formal. But first of all, think about the point of view of an individual neuron.

And we're going to add one extra input neuron. And symbolically, then we think about feeding a function, like a one-dimensional function, as inputs. So we're going to think of the output of this neuron, y of x, as just being the summation of scalar multiples of the input functions.

The observation of this slide is that if we add functions, at most, we add the number of kinks. And I'm just going to try to persuade you that that's the case with a picture. So here we're thinking of, the green function is our first function. And the blue function is our second function. And then the purple function shows the result of adding them together.

And notice that the green function has one kink. The blue function has two kinks. And then the purple function, which is their sum, has three kinks. And nothing more, you can't have more than that, because you only can have kinks at the places where the functions that you're adding had them.

So this is how we think about adding functions. We can at most add the number of kicks. But potentially the kicks could happen in the same place, in which case, it would be less than adding, because that wouldn't give you an extra one. But generally, if they're generally spaced out, then they would add.

Next intuition, the effect of applying ReLU. So we think now that we have a function f. And then we just take ReLU of f. So we think that the output is ReLU of f of x, basically. So y of x is ReLU of f of x. And now the claim is that if we apply ReLU, at most, we double the number of kinks.

And so this picture is supposed to illustrate. We think about the red function as being f. And then if we apply ReLU, we just chop off the positive part. And the observation is that if we had a linear region, that can at most get split up into two linear regions.

And if you just count on this picture, I think you see that the red function had 1, 2, 3, 4, 5 kinks, whereas the green function has 1, 2, 3, 4, 5, 6, 7, 8, 9, 9 kinks. So you can see it did nearly double in this case.

So now to be a bit more formal, we're going to imagine that we have a deep ReLU network with many layers. And we're going to examine one layer inside the network. And we're going to give a recursive definition of what this layer looks like. So the function at layer L is ReLU of the weight matrix times the function from the previous layer plus the bias.

And this is going to be an n-dimensional vector. The weight matrix is going to be an n by n matrix. This is going to be an n-dimensional vector. And the bias is also an n-dimensional vector. So that's the shape of all of these things.

And then we define this capital let KINKS-L to denote the maximum number of kinks over the n coordinates of the output. So remember, this is an n-dimensional vector. Each coordinate could have a certain number of kinks in its function, because each coordinate is a one-dimensional function.

And then we define this variable to be the maximum over those coordinates. And maybe I'm just going to write it as a claim. But the claim is that KINKS-L can be no greater than 2 times the width times the maximum number of kinks in the input.

So maybe you could think more about this. So the 2 comes from the doubling effect of the ReLU. And the n comes from the fact that you're adding n things. And then the trick is just to think about taking the max over each coordinate.

And then the other observation is that the kinks at the input, the input is basically 1, because there's no kinks. Does that make sense?

AUDIENCE: It should be 0.

JEREMY BERNSTEIN: It should be 0, you're right. But it's not going to-- Let's think about this after the lecture. But the point is that each time we can at most multiply by 2n. So I'm trying to get the answer that KINKS-L is no greater than 2n to the power L. OK, well, if KINKS-0 is 0, it's still less than or equal to 1.

[LAUGHTER]

Yeah, it looks like some accounting problem. But you see what the argument is. I just need to figure out-- Yeah?

AUDIENCE: You could start with one layer.

JEREMY BERNSTEIN: Yeah, you could just start with the base case being one layer. That's a great--

AUDIENCE: You should make it segmented instead. Otherwise, the ReLU adds a kink where there was none. Well, 0 to 1 is more [INAUDIBLE].

JEREMY BERNSTEIN: Yes. OK. Let's just talk more about this. But I think you get the intuition in making this rigorous. You just need to think about it a bit to work out what the base case should be. But I hope that the message gets across.

So in principle, we showed this statement. As a reminder, n is the width, capital L is the depth, and KINKS-L should be the maximum over the coordinates. And so the thing we wanted to point out is that the upper bound grows at most polynomially in width, but exponentially in depth.

So this is suggestive that there's a big benefit, if we're trying to maximize the number of kinks, in making the network deeper. Because every time we add a layer, we double things, rather than making it wider. So if our goal is to be able to approximate functions with many kinks, then it's better to make the network deeper rather than wider.

Does anyone have an objection to what we've shown? Because it is just an upper bound. So the objection that you could have is like, that's just an upper bound. Maybe it's never attained. Maybe in practice, when you actually build a very deep ReLU network, it's very difficult to get there to be a lot of kinks in the output function.

So the next step is to provide a construction to show you that, no, this can actually happen. And again, I'm just going to skip the slide. So again, the argument is that this g defined as a really small ReLU network in this form, if you plot what that actually amounts to, it amounts to a triangle.

And the property of a triangle is that if you iterate it, you apply it to itself-- so you do g composed with g, you actually get two triangles. There should be no gap there. Let's see, you get two triangles. And if you do g composed with g composed with g, you get four triangles. So you need to think a little bit to see why that is.

But actually, I think I didn't need to show you. But I was going to demonstrate it on the graphing website. But I think hopefully the claim is clear, that you can just with ReLUs, make a triangle function. And if you think about what happens to the triangle function when you compose it with itself, you just recursively get double as many triangles each time.

So the point is that this is an explicit construction of a certain ReLU network, where when you compose it with itself many, many times, you actually keep doubling the number of kinks in the function. So it can really happen, at least with this one example.

And then this is actually just running some numbers where I'm like, let's suppose that we take that triangle function and compose it with itself 500 times, that would give us at most 2 to the 500 kinks. No, it would actually give us that many kinks. It would really would give us that many.

Actually, each g turns out to have two layers. So this would be a 1,000-layer MLP. And then if you compute using the bound that we constructed, if you took a three-layer MLP, how wide would it need to be in order to fit the same function, you'd find that it would need to have almost 10 to the power 50 width in order to fit that same function.

So this is the point. This is the depth separation. There's a function, which is not even that difficult to write down. It's just a triangle or self-composed many, many times. And if you wanted to get a shallow network to exactly approximate that function, it would need to be very, very, very wide.

This is the reward for coming to the lecture, but this is also on the first problem set. So just maybe when you're solving the problem set, have a think about this. OK, I want to now think a bit more broadly. What does this not say?

This depth separation is suggesting that there could be a big advantage to making things really deep. But it's not telling us that deep networks are very easy to train, and they might not be. So it's not telling us about optimization.

And it's not telling us about generalization. It's not saying that the very deep networks would actually perform well on some particular data set that we want to apply them to. These are really just statements thinking about approximation. But this is not the end of the story. It's just one piece in this puzzle.

Again, there's more theorems of this nature. And feel free to look more into them. The one at the bottom is interesting. Because a question you could have is like, OK, if increasing depth is so great, maybe I'll just pick width 3 and just go really deep, and this will work really well.

But what that paper shows is that if the input space has dimension n, you need a width of at least n to approximate any function basically. So there can be a minimum width that you really need. Anyway, there's some further reading, in case you're interested to go deeper on this topic.

So in the last 15 minutes, I guess, I want to think about some more practical considerations. Those were the theoretical results. Maybe they're really interesting. But if you're actually building a machine learning system, I don't know if you need to think about those things. Maybe you do, which would be great.

But we're more just trying to expose you to these different styles of thinking. But now let's think a little bit about some more practical considerations. So the trouble is in practice, if you're actually building a machine learning system, we were saying that there's these three pieces of the puzzle. And isn't it nice if we can neatly tackle them one by one?

But out there in the real world, it's not like that. And basically, if you're training a system, you have all of these different issues conflated with each other. And if you've got a problem and you're training, you don't know if it's because your neural network actually can't approximate the thing that you're trying to approximate.

You don't know if it's because the training is failing. Interestingly, you would know if it was because it's not generalizing, because you could just compute the training error and the test error, and you can see if they're different. So that one's easier to diagnose. But the point is that all of these things are conflated.

And the other thing that we were going to say is, suppose that you work at a large language model startup and that your job is to make the LLM training as efficient as possible, then you actually really care about this question-- should I make the network wider or should I make it deeper?

For all those reasons that we talked about-- the computational cost of training, the computational cost of inference, in terms of making the model work well. In a sense, it's the most basic question that you're thinking about when you first learn about neural networks. But it's, to a large extent, an unsolved problem.

And I wanted to talk a little bit about this paper by Kaplan and McCandlish from 2020, which, this figure, people got T-shirts where they just had this figure, and then they would try to spread the word about. And this is the figure, which is basically saying as you make GPT bigger, it gets better.

And the point is that, in particular, the x-axis is compute, the amount of flops of compute, the size of your data set, and the number of parameters in the model. And the point is that the test loss, the y-axis, is always going down as you scale any of these axes. And the loss cannot go down forever because you can't have negative loss.

The loss has to go to 0. So it has to saturate eventually. But back in 2020, they were saying, hey, the models keep getting better. We should pay attention to this. Because then all of these advances in LLMs came. ChatGPT was created and so on.

But interestingly, they plot performance against parameters. In this top plot, it's not against width and it's not against depth. And the claim that they have in that paper is that within several orders of magnitude, the allocation of your compute budget between width and depth doesn't matter.

All that matters is the number of parameters and number of flops. You can have a depth 20 network or a depth 10 network with a commensurate number of width, it doesn't matter. So that's kind of interesting, that you just run the experiment and you see that width versus depth, within a large range, seems not to matter.

And they exemplify that with this lower plot. And the point is that once they subtract the embedding parameters-- so if we just focus in on the bottom-right plot, which is the model with the embedding layers not counted-- roughly you get very similar performance. I think what basically they're saying-- beyond depth 6, it doesn't matter what the width is or what the depth is.

The curves converge to each other. And all that matters is scaling up number of parameters. So this is just a very, very practical perspective, but from some very careful experimentalists about how things actually work in transformers. And you see it's basically saying-- within a large range, width versus depth doesn't matter; all you care about is parameters.

With that said, there's something I wanted to draw attention to, which is this idea of confounders. And there's another paper which people are also excited about, which people refer to as chinchilla scaling rules. The names keep getting stranger--

[LAUGHTER]

--for these things. But basically, they're interested in the same sort of questions as the previous paper I just showed you. But what they do is they question some of the results in that earlier paper. And one of the things that they suggest is that if you use a different learning rate schedule for the training, you can get slightly different, qualitatively different conclusions.

So if you just had one learning rate schedule and now you do a different learning rate schedule, you draw different conclusions. And this is just to say that precisely answering any of these questions is very difficult, because there are so many parts of the training pipeline that we don't understand.

So you may have one experimental setup and you say, hey, width versus depth doesn't matter. And then you change some detail about how you actually train the network. And now it's like oh, no, scaling width is a lot better because I fixed that issue. And I would call that a confounding issue, which is just some aspect of the training that we don't fully understand, but is having a bearing on the results.

So this is just what I want to point out, is that really resolving these questions experimentally is really hard, because there can be so many confounding variables. And resolving them theoretically is also really hard, because you have to work out how to think about these things. And a lot of these problems are unsolved because of the fact that it's kind of hard to figure things out.

So in conclusion, the summary. Oh, yeah, there's still another slide after this one. When you leave the lecture hall, try to do so quietly. Someone suggested that. OK, so the summary of the lecture is-- a very wide shallow neural net, so a three-layer MLP that's wide enough can fit any function within some class of functions, quite a broad class of functions.

Then, in the second half of the lecture, we said that deeper networks can fit certain kinds of functions with many fewer neurons because of their compositionality. And we called that a kind of depth separation result. So that was something interesting. And then the overall message that I am hoping to impart is basically, it's unclear how important these results are.

It's unclear how they interact with optimization and with generalization, but it's still really interesting to think about them and to have that as part of your toolkit. If you're thinking about your machine learning system, a basic question is, can the network architecture that I have actually approximate the function that I'm asking it to approximate? That's quite a basic question.

And I want to just conclude or to wrap up with a little preview for what's to come in the course. So I have a question. And it's, suppose that we have two machine learning problems-- one is an audio problem and one is an image classification problem.

So the first little picture is like a waveform. And then we're transcribing it to say the word "hello." And then the second picture is like a photograph of a person. And then we're classifying that as, that's a human.

And then the thought experiment is, in both cases, we could flatten those little pieces of data into vectors. OK, maybe the waveform is a vector. And then we can flatten that image of a person into another vector. And then we can just apply MLP to both.

We can take a multilayer perceptron with really nonlinearity and try to fit the first problem. And then we can take another MLP and try to fit the second problem. And the argument is, OK, the MLP is a universal function approximator, so this is a great idea. And so I want to ask, do people think that is a good idea?

AUDIENCE: No!

JEREMY BERNSTEIN: No? [LAUGHS] Why not? But why not?

AUDIENCE: You're not guaranteed to find the approximator. It exists, but you may not find it.

JEREMY BERNSTEIN: Yeah, that's one objection. That's a great point, that it could exist, but we might not find it.

AUDIENCE: I'd say efficiency, because although it is a universal function approximation in the [INAUDIBLE], it says you may or may not find that function in that space. To find it, you may need to tune certain parameters. It may take a while to find what you're looking for it. So after a certain cutoff, it may not be feasible.

JEREMY BERNSTEIN: The objection is it could be inefficient to use an MLP. There might be potentially another the architecture that could be more efficient. And in particular, it could be adapted to the type of data. Sorry?

AUDIENCE: I was just going to say, an MLP doesn't take advantage of the structure in the input data. So with an image, there's an inherent two-dimensional structure. And with a waveform, there's an inherent sequential structure.

JEREMY BERNSTEIN: That's a great comment. So the comment is that the two different problems have different structure in the data. And perhaps we want to adapt the model family. And that's what I thought when I was writing the slide. And then I was like, but wait a minute, now we're just applying transformers to everything.

So is that counter to that point? And then it was like, no, actually it's not. Because even if you apply a transformer to audio or you apply it to images, actually the preprocessing layer at the beginning of the network is very different. So if it's images, you have a special 2D patch kind of representation. And if it's audio, I assume they have some kind of better audio representation.

So even if you just throw transformers at everything, actually this comment that you do adapt the architecture a little bit based on what the data is. So yeah, I really agree with that comment. But it was a big question. People even recently have been thinking about that-- wait, are transformers all we need?

Maybe we don't need to model the data. But I think really, you still need to model the data little bit. And also transformers, you can ask how data hungry are they. Maybe there's a better model family which needs much less data in order to learn.

So that was the final thought to leave you with is, perhaps we want to match the architecture to the problem that we're trying to solve, or the structure of the data. Perhaps it could be more computationally efficient. Perhaps it could make it easier to find the approximator that we're looking for. OK, that's the end of the lecture. Thank you, everyone.