[SQUEAKING]

[RUSTLING]

[CLICKING]

JEREMY BERNSTEIN: I'm going to start. Thanks, everyone, for coming to the seventh lecture. So in the lecture, this is the plan. I'm going to introduce what the deep learning optimization problem looks like, which I think you already know. I want to cover some classical approaches towards optimization.

And these are things that, oftentimes, people in deep learning know about them and want them to work, but in practice, we don't use these things really to do deep learning optimization. So you want to introduce them, and how we think about them, and then think about maybe why we perhaps don't use them. And then towards the end of the lecture, I'll talk a bit about making the training really scalable and making the optimization algorithm scalable as you scale the architecture of your neural network.

And then at the end, I just want to talk a little bit about-- because this is of my research area-- also working on it with Phillip. So I want to just tell you a little bit about some of the things that we're thinking about. But you can be a little skeptical or ask the difficult questions, because that's what you should do.

So this is a reminder that, so far, we've looked at this puzzle. And I think we've looked at the first piece of the puzzle approximation and the third one, generalization. So these were the questions of does there exist a neural network in my model family that fits the training data? If it does exist, can I find it? And does it do well on unseen data?

But a perspective people have at the moment-- we could it may not be true, but just for the sake of argument. People are kind of like, approximation, we've kind of solved that now because you just use a transformer for everything-- really big transformer. Generalization we've kind of solved, because you just get so much data that everything generalizes.

If you have the whole universe of training data, you'll never overfit. And something-- well, probably those are slightly too extreme positions, but just for the sake of argument. And for the sake of argument, the thing that we can improve upon is optimization. And this is the mindset if you're at a deep learning startup and you just want to make the optimization as efficient as possible, and then you just throw the big neural network at the problem and collect lots of data. But maybe two-- what's the word-- caricature of a thing.

But anyway, this lecture is just going to talk about the second question optimization. So I wanted to write it down in a formal way, where we have a neural network, which is a function f. And it takes inputs and weights, and it gives you an output. And we have an error measure, which I'll call L. And you think of this as cross-entropy or square error. And you think of y hat as your prediction and y as your target. And it somehow just measures the difference between them in some way.

And then we collect here training data. So here, I'm thinking about paths of data with inputs and targets. And capital N is the number of data points in my data set. And then I'm going to write down an example of a classic machine learning or deep learning loss function, which is something like the average over the data set of-- let me just call it Li to be the error on the i-th data point.

And Li would be-- what that means-- it's shorthand for the loss between the output of the neural net on example i with weight w. And it's the discrepancy between that and the i-th target. So this is kind of what a loss function oftentimes looks like in machine learning.

And you pick f-- these days, we pick f to be a really big transformer. N, we make it really, really large. And then we just try to find the W that minimizes this loss function. So there's two features I want to point out at this stage. One is that the loss function has a kind of compositional structure, where we have an error measure and a neural network, and we glue them together.

So there's two things that-- the error measure in the neural network and that composed with each other. So that's an interesting part of the loss function. And the other is that we average over the training set. And this is a picture of what we're doing. So we set up this loss function. And we start at some weight setting that has a large loss. And then we try to move the weights to the place with the small loss. And we think about going downhill on this. And we compute the gradients. And we try to get closer and closer, and eventually stop at this point hopefully. And so that's just saying that we iterate this gradient descent operation.

Oops. Let's see. And this is the learning rate or the step size, which I just call eta. And this is the gradient. So what makes this hard-- in a sense, it's not hard, because we do it all the time, and we can train really big machine learning models. And it doesn't seem to be-- we can do it. But what are some of the things which make this like challenging in a way-- some of the examples are There are lots of weights. It's really high-dimensional optimization.

Another thing is the neural network may be very deep. It has lots of layers, and that can make-- for a while, we couldn't train really, really deep neural networks. But with some advances, now we can. What else is hard? There's lots of data. So usually, we do stochastic gradients. We compute gradients on a handful of data points. And then on the next iteration, we pick another handful of data points. So that means that we just have noisy estimates of the full gradient across all the data points.

For the purpose of this lecture, I actually kind of want to ignore this problem. And these two first problems are already kind of-- if you just think about those ones, there's really a lot of interesting things to say and things to think about. So in other words, for the sake of this lecture, let's just assume that we're in the full batch setting, i.e., we can evaluate the true gradient of the whole loss function on all the data at every step. Let's just assume that we can do that.

So towards the end of the lecture, I want to talk a little bit about some of the problems that we have when we do scaling-- when we scale up or scale up our neural network. So I want you to think that this is a neural network where I pass in an input here, and I get the output. And I'm either going to be interested in scaling the width, like making the network wider, or potentially scaling the depth and making the network deeper.

And we kind of run into two nuisances when we do this if you do it naively. And the first one is actually a little bit explored in this homework. So this is a plot of training loss. And I'm training networks with a range of learning rates on the x-axis. And the different curves are increasing width. So as I go down, it's getting wider.

And what you typically see is that if you make your neural network wider, you see that the best loss you achieve, you see it kind of-- you get a better loss as you make the network wider, which is good. That's why people want to scale their network, because it improves the performance.

But you see that on the other hand, if you look at the learning rate, which gets the best loss, it changes. As I scale my model, the best learning rate changes. So that means like, typically, if you train a neural network, you don't run this full sweep of learning rate. You just pick a learning rate, and then maybe you try to scale your model, and you try to run the training again. And you find, oh, my training isn't working. Because just because I made my neural network bigger. But then you have to retune the learning rate at the large scale.

So this is just like a kind of nuisance that, as we scale the whole lost kind of landscape as a function of step size is kind of drifting. That's a nuisance. And similarly, if we scale depth, and we make the network deeper-- so it's the same. The different curves here are, as I go down it's increasing depth. And we're plotting training loss against learning rate. And something you can see-- if you do this naively, you just find that performance gets worse as I make my model deeper, which is-- that's undesirable. Yeah?

AUDIENCE: Isn't that the optimal learning rate? Because it looks like the training loss kind of drifts, but it maintains the same shape. So the optimal learning rate and finding that minimum be the same [INAUDIBLE] loss exists.

JEREMY BERNSTEIN: Yeah, let's just pick two points and just compare them. So let's say that we're at width 32. The best loss is about here on the dark blue curve. And that corresponds to this learning rate, which is if we're at a larger width, 1,024, the best loss occurs here. And it just corresponds to a different learning rate. So it's the fact that the minima occur at different points on the x-axis. That's what I'm trying to point out. Yeah?

AUDIENCE: Do we have any intuition of why the optimal learning rate changes as we scale the width.

JEREMY BERNSTEIN: Yeah, that's at the end-- towards the end of the lecture, I'll try to explain why. Oh, yeah?

AUDIENCE: So I'm just curious-- in practice, people use deeper neural networks. But if you're just looking at this graph here, you would not use one because the loss is worse if [INAUDIBLE]. So how do you reconcile?

JEREMY BERNSTEIN: Yeah, so this is basically pre, let's say, 2015. The max depth people went to was, let's say, 16 or something. And then people figured out techniques. And now, you can train a network with thousands of layers. So I'm just saying if you do things very naively-- if you don't know that much about deep learning, this is like-- you would see this type of behavior. Then, as you learn more and more, then you know how to fix these problems. So I'll talk a little bit about how to fix some of these problems.

AUDIENCE: [INAUDIBLE] with those fixes, that you can bring loss back down [INAUDIBLE].

JEREMY BERNSTEIN: Yes, essentially, the fix for depth is use a residual architecture and set up the residual blocks in a good way. And you could use a residual architecture and set up the residual blocks badly, and you'd still see this kind of behavior, or set up the interaction between the optimization algorithm and the residual blocks. So if you solve at least enough of the problems, you can fix this behavior. And if you solve all the problems, then you can fix it in a really nice way where everything-- the optimal learning rate always transfers across scale and so on.

AUDIENCE: And that's the same on the left where you can eliminate the drift?

JEREMY BERNSTEIN: Yeah. The point on the left is that even with all the latest fixes prior to 2021, it had that drift. But then if you know all the literature from 2021 until now, then you can fix it for sure. So anyway, these are just some examples of some problems how things can go badly when we scale.

And if you're trying to scale massive transformers, you may not even have the resources to tune all the hyperparameters of a really big model. So you may be like in this regime where you can only tune things at small scale and then try to transfer them. So that's one of the applications of fixing this type of problem.

That was just a kind of teaser for the end of the lecture. Now, I want to just-- let's go back to basics and just how should we think about designing optimization algorithms and solving optimization problems? So I want to look at a couple of different types of methods. The first is first order methods, which only use first derivatives. So they basically only use gradients, which if you think about it, is kind of what we do in deep learning.

And I'm going to use this symbol g as a shorthand for the derivative of the loss with respect to its weights. So g is always going to mean the gradient of the loss with respect to the weights, i.e., the thing that you get by doing back propagation. And then there's also second order methods which, in addition to first derivatives, they also use the second derivatives of the loss function. So they're called second order methods.

And this symbol h is going to be a shorthand for the second derivatives of the loss with respect to the weights. So it's a big matrix of second derivatives. So you can think of the gradient as being one big vector. The Hessian is a matrix of second derivatives corresponding to all pairs of second derivatives.

And the thing that I hope to get at is to explain the modeling assumptions that different approaches make and then also think about potential shortcomings of those modeling assumptions. So you should always ask-- if I present something to you, you should always say, well, why don't we do this? Why don't we actually do this in practice? That should be the question that you have.

So all of these different methods, I would argue, start by Taylor expanding the loss function. So different classical approaches to optimization take the Taylor expansion as a starting point. And I'm writing this Taylor expansion like this. But because we introduce the shorthand, I can rewrite it like this. So the first order term is gradient inner product delta w. And then the second order term is-- it's this Hessian term.

So the Taylor expansion up to second order is like a quadratic form. And then it's like a multivariate Taylor expansion. So again, this is the loss evaluated at a kind of perturbed weight vector, or weight vector plus a delta w. And then we can just write out the Taylor expansion. And I want to give-- these two terms, I want to give them a name. And we'll call them the linearization of our loss function.

And then all the higher order things we'll call the nonlinear part. And again, we're using this shorthand. The gradient is this, and the Hessian matrix is this thing. And I just wanted to just visually give you a picture. So the gradient we can think of as this big vector. And the Hessian we can think of as a big matrix.

If there are d parameters in the whole neural network-- the size of the weight space is like d dimensions, the gradient is d by 1, and the Hessian is d by d. So it's a d by d matrix. So our first method that I want to talk about is called Newton's method.

Do people already know-- is it something that people have heard about? Well, let's just quickly talk about it. So the idea of Newton's method is we kind of drop-- or remember there were all those higher order terms in the Taylor expansion. We just kind of forget about them. Let's just ignore them and take the Taylor expansion up to first order.

And then we're going to think about picking our optimization step as choosing a delta w. We think of w as our current weights. And we're going to pick a delta w to add to them. And the way we're going to do that is we're going to minimize this quadratic form with respect to the weight perturbation delta w. Wait, does anyone know how do I minimize that quadratic form? Can someone tell me how do I do it?

AUDIENCE: Take the derivative of [INAUDIBLE].

JEREMY BERNSTEIN: Yeah--

AUDIENCE: That's what [INAUDIBLE].

JEREMY BERNSTEIN: Exactly. So we differentiate it with respect to delta w which gives me lambda over 2 times 2, h delta w. And then these cancel. And then we set the derivative to 0. That's that thing that we all did once upon-- yeah. And then we just solve for delta w by rearranging, and we get like this thing.

And we just call this Newton's method. And people describe this as you get your gradient. And then they talk about preconditioning it with the inverse Hessian. But it basically means you take the Hessian, you invert it, and then you multiply it with the gradient. And we call this Newton's method for-- it's an optimization algorithm that uses first and second order information. So I just rewrote that here. So what are the problems? Why don't we actually use Newton's method to train neural networks? What are the--

AUDIENCE: [INAUDIBLE]

JEREMY BERNSTEIN: Yeah. So problem one is that the Hessian is d by d matrix. And d may be billions in a large neural network, so you can't even store such a large matrix. And if you could, you could invert it. And you just basically-- it's just too big. Another problem with Newton's method is, if you think about how we derived it-- are we sure that this procedure is giving us a minimum of the quadratic form? Do we know-- all we did is--

AUDIENCE: It's one level.

JEREMY BERNSTEIN: It's just the first-- yeah, so what do you mean? Could it be a minimum, or could it be something else? It could be a maximum. It's just finding a critical point of the quadratic form. If it's close to a max, it could find a maximum. So it's not even necessarily doing gradient descent. It could be doing gradient ascent.

And there are ways to fix these type of things. There's something called cubic regularization, which adds a cubic penalty to the quadratic form. But people don't use that. And any time we present a method, there's always like attempts to make it practical. And like there's a literature on that method. But then in practice, we just use Adam to train neural networks.

So that's Newton's method. Now, we're going to look at something called Gauss-Newton method, which is a little bit different. And it starts with something called the Gauss-Newton decomposition. So here, we suppose that we have a composite objective function. It's the composition of two functions, which is-- remember, we actually have that in machine learning, because we have an error, and we have a neural net.

So we're actually in this situation. That's what this is saying. So let's derive the Gauss-Newton decomposition. So we start with the gradient, which is dL by dw. Because we have a composite we can do the chain rule. So we get dL by df, df by dw. So that's just the chain rule.

And now, let's say we want to know about the Hessian. This is the second derivative of the loss with respect to the weights. Well, now, we just need to differentiate the gradient again. But we recognize now that we have a product. So we can do like the product rule. And let me just remember how to do the product rule.

d-- so this is like the derivative of this with respect to w, and then one application of the chain rule. And then there's this term, which is just like this term. And I haven't differentiated it. And then we have to do the second thing, which is-- like, I don't differentiate this one, and I do differentiate this one. So I get dl by df, D squared f by dw squared.

And so what I hope to have shown you is that if you just have a composition of two things, you can derive a formula for the second derivative, and it splits up into two pieces because of the product rule. And we just call that the Gauss-Newton decomposition. So I wrote it out. So what this is saying is that the full Hessian of a composite decomposes into these two pieces.

And we think if we think of this as being the curvature, it's like the second order information about the whole objective function, you can break it up into the second order information about the error with respect to the model, and the second order information about the model with respect to the weights. So it's like decomposing curvature into one piece from the error and one piece from the model. That's called Gauss-Newton decomposition.

So then what people do is they say, let's pretend that our error measure is the squared error. And we know that for the squared error, the second derivative with respect to the model is just basically one or some-- yeah. So then, basically, we can drop this term. We just say that that's 1.

And then they say, let's just ignore the curvature of our model, and let's just ignore this term. It's just an assumption that people make. They say, hey, I don't really know how to deal with this term, so let's just ignore it. So under those like steps, h which is d squared, l by dw squared, is just equal to somehow the product of df by dw and df by dw.

So then, if I want to do a Newton method but using this form of the Hessian, what does Newton's method look like? It's delta w is minus h inverse g, which is now minus df by dw df by dw inverse g. If you go through this, you'll realize that all of these things are tensors and you need to be careful about all the indexing. But that's something to just check.

And we call this thing the Gauss-Newton, method because it uses-- it started with the Gauss-Newton decomposition. And it's just another way of deriving an algorithm which looks some kind of iterative weight update.

And I've just rewritten it here. So this is thing that we're calling the Gauss-Newton method. And remember, these are the derivatives of the model with respect to the weights. So it's not the same thing as the regular gradient that you usually think about. It's a different gradient.

And this is just to remind you about the assumptions that the method made. One is just drop this term, and the other one is like set this one to 1. So what do people think of-- what's the problem? Why don't I actually do this?

AUDIENCE: For deep learning, you may have a low rank Gaussian approximation.

JEREMY BERNSTEIN: Oh, so that could be a reason why I would want to do it. Is that what you're saying?

AUDIENCE: No, why you wouldn't want to.

JEREMY BERNSTEIN: Oh, because you're saying if it's low rank, I can't invert it? Yeah, I need to think more about that, but you might be right. Oftentimes, people will always just-- if they want to invert something, they'll add a little bit of identity matrix to it to give it better conditioning, and then they'll just invert that.

Yeah, that's a great point. Basically, what I wanted to say is it involves inverting a matrix. It involves forming an extra matrix and then inverting it. And not only that-- it also involves computing these extra derivatives. Not only do you have to compute the regular gradient, which is this one. You have to compute these extra derivatives.

And people are not going to want to do that in practice unless they're really convinced that there's a really big improvement to using this method. And basically, nobody's convinced them of that, so they just don't use it. That's the kind of practical reason why people would not use this. But there's a lot of research trying to make this type of thing practical. So I'll just write like extra gradients that you've got to compute, and then inverting matrices. These are the kind of things which count against it a little bit. Yeah, go ahead.

AUDIENCE: I just don't get why you have to compute gradients. Isn't the partial derivative of [INAUDIBLE]?

JEREMY BERNSTEIN: It is, but because you do the gradient by backpropagation, you never explicitly form df by dw. If you think about how you actually calculate g, you'll realize because you start from the end of the network and work backwards, you never compute df by dw. But it is implicitly there, but you're not computing it. But if you did forward mode automatic differentiation, then I think you would get both of them. But it's much more expensive to do forward mode automatic differentiation.

So that was Gauss-Newton method. This one is called steepest descent. So now we're in-- I'm not sure whether you should think of Gauss-Newton as being first order or second order because-- oh, yeah?

AUDIENCE: I was wondering, could you give us an intuitive for how much more expensive the inversion would be or the gradient of [INAUDIBLE]? Because when I hear [INAUDIBLE] matrix, I'm like, well, we're already training a neural network, so how much [INAUDIBLE] would it be?

JEREMY BERNSTEIN: Yes, for Gauss-Newton, I'm not-- let's just talk about Newton, so let's pretend it's the full Hessian. I'm not an expert on all these. And I think isn't it cubic-- the cost of inverting a matrix is basically equivalent to computing a singular value decomposition? And I just have an intuitive sense that you never want to do those. They're much more expensive than computing a forward pass or a backward pass. That's the--

AUDIENCE: Are forward passes and backward passes quadratic or linear in the input?

JEREMY BERNSTEIN: In the-- so how do they scale with the dimension? That's the question. Well, they're somehow linear in the number of layers. And then you need to ask what is the cost of doing a forward on an individual layer. And then you also have factors-- this is a good thing to think about. I'm not going to give you the answer because I don't have it.

But the other factors that are involved are GPUs are great at doing matrix multiplication. They're basically designed to do forwards on layers and to do backwards on layers, but they're not designed to invert matrices or do singular value decomposition. So these are things that, if you just get a Colab notebook and start playing around, very quickly, you can get a sense of how these different algorithms compare against each other in terms of complexity. But these are the right-- yeah, they're the right questions to ask or the right things to think about. Sorry.

AUDIENCE: For the last slide where you mentioned that we need to compute the partial f, partial w explicitly, and we're not doing that from a backward pass, is it because for backward pass, we're actually computing the gradient for each module, like a sub-part of all parameters? But here, you really need a big matrix including the gradient [INAUDIBLE] with respect to all the parameters in one matrix?

JEREMY BERNSTEIN: Yeah, let me just give you an intuitive sense of why we're definitely not computing those derivatives doing a regular backward pass with that. Maybe that would answer the question. But think g is the derivative of the loss, which is a single number with respect to all of the weights, whereas the output of the network, it could be outputting a tensor.

And df by dw is, for every component of that tensor, it's 1 derivative. It's d-- the d-- sorry, it's the derivative of the first component of that tensor with respect to all the weights. It's the derivative of the second component of that tensor with respect to all the weights. So it's just a bigger tensor than the gradient.

And you really want to avoid explicitly forming that. And backpropagation allows you to do that, because you only ever track derivatives with respect to the loss, which is a single number. Yeah thinking through-- just actually implement backprop for yourself is a really good exercise that forces you to-- it forces you to think about these things. And because PyTorch and because the packages we have are so automated, you can actually have a whole career without ever implementing backprop or thinking about what gradients look like. But it's a really good exercise to do that, I would just say.

I think I'm doing OK on time, so yeah, don't be afraid if you do have questions. So the next one is called steepest descent. And again, we get this Taylor expansion. So always get this Taylor expansion. And this time, we're going to-- again, we're just going to throw away the nonlinear part, and we're just going to replace it with lambda over 2.

So lambda is a number. It could be 5. And then we choose our favorite norm. And we measure the norm of delta W And we square it. So steepest descent-- what I'm saying-- is a technique where you just-- you say the nonlinear part-- I don't even know what that nonlinear part is. Let me just replace it with something which I know very well. And actually, a lot of optimization methods actually have that flavor of just take the thing that I don't even know how to deal with it, and just replace it with something that I like. Yeah?

AUDIENCE: But what's the rationale behind this replacement? Will you choose this specific replacement or something else?

JEREMY BERNSTEIN: That's a great question. I'll try to answer it, but let me just first show you what the implications are of different choices. And then we can try to think about how we would actually make this choice. So the point is, what are some of people's favorite norms? I'm going to start-- Euclidean norm. Someone else?

AUDIENCE: Frobenius.

JEREMY BERNSTEIN: Frobenius. So that's if we have a matrix-- if we have a weight matrix, Frobenius norm. What's another one?

AUDIENCE: Max.

JEREMY BERNSTEIN: Max, the infinity norm. Is that what you meant? Yeah. One more.

AUDIENCE: LP norm.

JEREMY BERNSTEIN: Pardon?

AUDIENCE: LP.

JEREMY BERNSTEIN: Say--

AUDIENCE: LP norm.

JEREMY BERNSTEIN: The LP norm?

AUDIENCE: Yeah.

JEREMY BERNSTEIN: Yeah, which P equals 2 is Euclidean. One-- some of the-- yeah.

AUDIENCE: Is the KL divergence kind of different--

JEREMY BERNSTEIN: Yeah, it's actually not a norm, but let's just add it. It doesn't satisfy all the properties. It's not symmetric. It's not something. So it's actually not a norm, but it's another-- that's a measure of distance, but it's technically not a norm.

So the point to illustrate is there's a lot of norms. And so there's a lot of steepest descent methods because there's a lot of norms. And there's many more norms than what we talked about. So let's just start with L2 norm or Euclidean norm. Now, we're considering this a model because we threw away what our loss function actually is, and we're just modeling the nonlinear part.

And just to remind you that a ball of fixed Euclidean norm is a circle or a sphere. These are two balls of fixed Euclidean norm. And we're going to do our favorite thing of minimizing the right-hand side. How do we minimize it? Can someone tell me how do I minimize it?

AUDIENCE: You could [INAUDIBLE].

JEREMY BERNSTEIN: Take the derivative, set it to 0. And it's g plus-- you need to work out how to take a derivative of a squared Euclidean norm. But it's just lambda delta w is equal to 0. And I rearrange it, and I get delta w is minus 1 over lambda times the gradient.

And so then I recognize this. I say, hey, this is just the most vanilla form of gradient descent with a particular step size-- like, a step size of 1 over lambda. I make the penalty larger by increasing lambda, and my step size gets smaller. So it's like, it's kind of interesting that there's some connection between regular old gradient descent and L2 geometry on your optimization space.

So let's do another example. So this time, we put the infinity norm. And remember, the unit ball in the infinity norm is not really looking like a ball anymore. It's more of a kind of square. So infinity norm is just measuring the maximum size of a vector across the coordinates. I can just write that.

And the absolute value. So has anyone done the homework already? Does anyone know the answer? If I minimize the right-hand side, what is it? Does anyone know?

AUDIENCE: It's [INAUDIBLE] 1 over lambda [INAUDIBLE] max something.

JEREMY BERNSTEIN: Yeah-- see, this is on the homework. So if you just solve this-- it's not quite as simple as just differentiating because of the-- because a max is not really continuous in some sense. But the solution is like the L1 norm of the gradient divided by lambda times the sine of the gradient, where the sine is plus or minus 1.

It's plus 1 if the thing is positive, and minus 1 if the thing is negative and you apply that across the coordinates of the vector. So it's like, oh, I set my norm to infinity norm. And now, I get this sine gradient descent algorithm. Interesting. And there's a reading that we posted, and it's a blog post that talks a little bit about this, if you want. And it has some interactive visualizations if you want to visualize some of these things.

And I still want to get-- that question of how should you choose a norm in practice is the right question to ask, but we still haven't answered that question. I'm just showing if you pick this norm, you get this algorithm. If you pick this norm, you get this other algorithm. The question we should ask is, how do we pick such a norm?

But before that-- and I guess-- yeah, let's just look at how this looks for a general norm. And even though it's a general norm, we can still actually say something about it. And what you can show, and I think this is also on the homework, is to show that it-- the solution to this thing can be transformed into this other thing.

And we can think of this as like-- we think of this as being the step size or the learning rate. And we think of this as being the direction-- say, the step direction. And we can separate the solution of the steepest descent problem into two pieces. One is computing the step size. The other is computing the direction.

And this funny thing is called the dual norm. Every norm has a dual, which we call it dual norm, and it appears in this-- anyway, this is on the homework. And this is something that you can show. So there's a general formulation of the solution to the problem. Yeah, go ahead?

AUDIENCE: I see the [INAUDIBLE], but there's-- I mean, [INAUDIBLE]. When we arrive at a certain delta w, there is no guarantee that the above approximation still holds when we plug in the delta w [INAUDIBLE].

JEREMY BERNSTEIN: There's no guarantee that--

AUDIENCE: That the approximation either the second or the third--

JEREMY BERNSTEIN: Yes, there is no guarantee. Yeah, there's no guarantee about this at all, because I said take your loss function, throw away the nonlinear part, and just replace it with your favorite norm. So that doesn't sound like the kind of thing that's going to give you a guarantee. It sounds kind of random. But it's a framework for thinking about deriving different optimization algorithms. For the framework to actually apply, what you would want to do is you would want to-- yeah, let's just talk about this because of the fact that we can--

AUDIENCE: In a sense, we control how much we deviate from the actual weights, right?

JEREMY BERNSTEIN: Yes, the larger and larger lambda is, somehow the better the model should be, in a sense. Is that what you're saying? Because yeah, you never move too far. If you make lambda infinite, you never move anywhere, so you're always within the linear region, and then the model holds. But it's like, you wouldn't know how big you need to make lambda to be for that to work. That's the problem. Exactly.

Let's just think a little bit about how we could really pick such a norm. So you remember that-- let's just go back to get the Taylor expansion. So this is the Taylor expansion. Assuming that the loss function is an analytic-- which it's not always, but let's assume it is-- then this Taylor expansion really holds.

But what if we could do the following thing? What if we could upper-bound the Taylor expansion with the linear piece, and we could produce a lambda and we could produce a norm such that this was really an upper bound? If we could produce such a lambda, then we could get the kind of guarantees that you're talking about.

And actually, on the homework, you're going to do this. Although it's a bonus question, so maybe you're not going to do it. But if you want to, you can do it. And for the case of a linear model a linear predictor in the square loss. And then the hope of putting that on the homework is that you would then think, what if I don't have a linear predictor? What if I have a two-layer neural net? How would I do it then? What if I had a five-layer neural net? How would I do it then? And that's essentially what I'm trying to do in my research at the moment, is try to answer that question.

Anyway, so that's the end of classical methods for optimization. And now, I want to think about, on a purely heuristic and purely intuitive level, how to make the training really scalable. And then if there's time at the end, which it seems like there might be, we can start trying to think about how to build a more formal theory around this type of thing. Does anybody have more questions?

AUDIENCE: So for the infinity norm, do we update all the components of w at the same time?

JEREMY BERNSTEIN: Yeah--

AUDIENCE: So with the same kind of magnitude of [INAUDIBLE].

JEREMY BERNSTEIN: Exactly. That's why the sine function appears, because the sine of a vector maps a vector to another vector, where all the components have the same magnitude.

AUDIENCE: Can you just repeat the questions for the--

JEREMY BERNSTEIN: Oh, sorry. To repeat the question, it was a question about, does the solution to steepest descent have the same magnitude update across all the different coordinates? And the answer was yes.

AUDIENCE: Sorry, can we go-- for the general steepest descent-- because you decompose into step size and step direction. And for the step direction, if you look at the dot product, then the direction we're going to maximize is exactly the gradient direction. I just don't know what's the intuition behind why imposing the-- imposing a constraint on the norm of your self-direction is going to help.

JEREMY BERNSTEIN: So if the constraint is the-- sorry. The question was, under a Euclidean norm penalty, the solution is to always point the step direction in the same direction as the gradient. And that's kind of an intuitive thing. But what seems less intuitive is if I put a different norm penalty. The optimal direction may change, and it may point in a different direction from the gradient. And the question is why? Why would that ever be a good thing to do?

AUDIENCE: Yeah, [INAUDIBLE] want to consider a norm like this? Why would I want to impose a norm constraint on my step size direction?

JEREMY BERNSTEIN: That's exactly the question-- that's the right question. Why would I want to do that? Why would I want to impose a different norm other than the Euclidean norm? So I'm trying to say that, implicitly, by-- I'm trying to put it upon you that, implicitly, when you say the best direction is the gradient direction, I'm trying to say that you're thinking Euclidean.

So the question-- why would another norm be better, which would tell you to go in a different direction? Well, the rough intuition is, what if all the axes in my space are not equal with each other? Imagine you get a map of the US, and then you take the map and you squeeze it.

So you get the corner in Photoshop, and you squeeze the corners together so it becomes very-- and then you-- so taking a step of 1 centimeter East-West would correspond to moving like 10,000 miles, 2,000 miles, whereas taking one centimeter step North-South might correspond to going 10 miles just because you stretch the map in a weird way.

And when you say that it's natural that going in the same direction as the gradient is the right thing to do, you're assuming that you have a very isotropic map of the United States, where taking steps of equal-- and what I'm trying to-- the idea is that you might not be so lucky that the loss function-- because of-- in deep learning, because of the architecture of the network and so on, it may not have that property that different directions in the weight space are equal to each other-- may be a very non-isotropic space.

AUDIENCE: So you're saying that there's actually some-- doing different norms somehow equivalent. I'm scaling each coordinate-- I'm basically using a preconditioner matrix to scale each coordinate at a different rate?

JEREMY BERNSTEIN: That's one example. Yes, so like that-- you could take a norm-- I can say that I'm going to build this new norm. So remember the Euclidean norm is like sum over i wi squared. And I could say, well, I want to build a new norm, where I take positive constants, and I just scale the different coordinates differently.

I could do that. And I could derive a steepest descent algorithm under this new norm, and it would correspond to some reshaping of the gradient. But that's just one example. And another example is the infinity norm, or the Frobenius norm, or the-- and they're all ways of measuring-- of saying that different coordinate directions or different directions more generally have different weightings.

And maybe one more thing to say about that is why is it different weightings have different-- another intuition is if I've used the linear piece, like the first order term, I need to know for how far that linear piece is valid. Maybe if I move too far in that direction, the linear piece broke down. And it may break down at a different rate going in different directions. So the idea of building a norm is to capture at what rate does the linear piece break down if I move in this direction or this direction. It may not be the same.

So now, we're going to forget that for a little bit and just thinking about scaling. So the intuitive question that I want to ask is, how large should the weight updates be when I do gradient descent? And I want to think about this kind of Goldilocks step size. Because if it's too big, intuitively, you're going to break your neural network. And if it's too small, you're not going to be training very efficiently because your steps are too small.

So you want to find that nice middle. And someone already had this observation. The observation is that a neural network is built out of matrices. It's not one big vector. The weight vector is like matrix 1, matrix 2, matrix 3. So can we think about that matrix structure? And this is where I wanted to say, perhaps we could try a matrix norm. And the one example was the Frobenius norm.

So the idea is-- oh, yeah, I forgot to say. Whenever anyone talks about too big or too small, you should always ask in which norm? It's analogous to if someone just says something's really big, you have no idea what they're talking about, because you always need some kind of reference, like a scale. And if you look at a map, there's the scale at the bottom, which says five centimeters corresponds to 1 mile or something.

So if anyone ever says that something is big or small, you should always ask, what's the scale that you're measuring it on? And for tensors, the notion of scale is what's the norm. So that's a good thing to think about. So in particular, when we say that we want the update to not be too big or too small, we want to choose a norm, and say it should not be too big or too small in that norm. And then we're going to say a neural network is built out of weight matrices. Maybe we can try a matrix norm. Does anyone know any other matrix norms?

AUDIENCE: But you can also take the largest singular value that's also a norm.

JEREMY BERNSTEIN: Exactly. We call it the spectral norm, or it has also some other names. Does anyone else know any other ones? Just wait. In fact, I think I pretty much knew just the Frobenius norm until maybe three years ago. So spectral norm is the largest singular value.

There's a bunch of other ones. One is called the nuclear norm. There's a general class of norms called the Schatten p-norms. And just wanting to point out, there's actually a lot of ways to measure how big a matrix is, which kind of makes sense because it's got a lot of different coordinates. So there's presumably a lot of different ways to measure size of a matrix.

This is where I want to introduce-- so just to summarize, we want to ask how big should our weight update be? And we've said, well, we kind of need to pick a norm in order for that question to be a meaningful question. And now, I want to introduce you to this new perspective in thinking about a neural network.

So remember that maybe when you first think about a neural network, you think about neurons, and there's nodes, and they're connected by edges. And then you go a bit further, and you learn a bit more. And then you're like, oh, I recognize this. This is a weight matrix. And if I have a layer with a single output, then its weights are just a vector.

And now, I'd start to describe my neural network as like matrix multiplications. And I think of this as the tensor perspective. And then what I want to say is you go even further, and you realize that there's this kind of spectral perspective on neural networks, where every weight matrix has something called a Singular Value Decomposition, or an SVD, which is like a generalized eigenvalue decomposition.

Not every matrix has eigenvalue decomposition, but every matrix has a singular value decomposition. And we can think about taking my neural network and decomposing each matrix into its SVD. I'm not saying we should actually go ahead and do that to train it in its spectral representation or something like that. I'm just saying that mentally, you can always do that. It always exists, that decomposition.

I just wanted to introduce-- a way that I think about stable training is I kind of imagine my spectral decomposition of all my weight matrices. And one thing I think is that probably the singular values should not change too drastically from step to step. That would be bad. But if they change too little from step to step, that would be also bad because I would be training too slow.

It's like, another way to think about what stable training means is, maybe you don't want your spectral decomposition to be going haywire when you're training, and maybe you don't want it to just change a teeny tiny bit-- something in the middle seems good. That's a thought. And then as somebody mentioned, there's the spectral norm.

So if I want to compute, let's say the spectral norm of this weight matrix, I compute its singular value decomposition, and then I just take the largest singular value. And that's like a way of computing the spectral norm. So that's another way of measuring the size of a matrix, which you'll also come across in the homework.

So I wanted to talk a little bit about the spectral norm. How do we think about this thing? And I drew this picture. And the idea is the spectral norm tells if I feed a vector into a matrix, and I get something that comes out, it tells me how big the thing that comes out can be given that I knew how big the thing that went in was.

So we can write that. I'm going to write it. So I'll use this notation. The spectral norm of a matrix M is the max over input vectors with unit L2 norm of the L2 norm of MV. So does this make sense? It says, I feed in a unit vector to the matrix, and I get out a vector. How large can that vector possibly be? That's the spectral norm.

I think that that's a nice idea, because when you train a neural network, you're putting vectors into matrices, and matrices are coming out the other end. Actually, something about this norm is descriptive of what's actually happening during training. So maybe that's a good potentially that could be a good norm for us to think about when we measure the size of the matrices. That's the idea. And so there's a fact, which I have not proved, but the spectral norm is equivalent to the largest singular value of the weight matrix.

But there's something arbitrary here. And what I hope to point out as arbitrary is, why would we care about the L2 norm of the inputs? That was an arbitrary decision. And why would we care about the L2 norm of the outputs? That also seems arbitrary. We can actually change this to be arbitrary, let's say, LP norm here and L2 norm there. And you can change this definition to involve any norm on the input space and any norm on the output space.

And then we would refer to the thing that we get as the, let's say, the q to p norm on the matrix. It's called inducing-- we call this inducing a norm on a matrix given a norm on the input space and a norm on the output space. So we call this an induced operator norm.

And this seems kind of helpful, because in my neural net, maybe I don't care about the L2 norm of my activations, and maybe I don't care about the L2 norm of the outputs of a layer. Maybe I care about some other norm. And that's where we come to this thing that we're going to call the RMS to RMS operator norm. I'm not sure if anyone remembers about the RMS norm? Does anyone remember what RMS norm is from my other lecture? It's related to the L2 norm. Does anyone remember? It was on one of the--

AUDIENCE: The norm scaled by the number of coordinates.

JEREMY BERNSTEIN: Exactly. Yeah, so we define this RMS norm of a vector v to be the 1 over square root d, if d is the dimension, times the 2 norm of the vector. So it's just a rescaled Euclidean norm. But does anyone know about-- if you know much about transformers or neural architectures that people use, we often make a big effort to normalize all of the activations of the layer in the RMS norm.

And this is referred to either as RMS normalization, or layer normalization, or layer norm. It's a standard thing that, if you actually implement a transformer-- I think at some point you'll do this-- you'll probably use layer norm. So it's like I want to make the argument that for a layer in a neural network, a really natural norm to equip it with for an activation vector is the norm, not the Euclidean norm.

And another way to put this is what does it mean if the norm of a vector is 1? If v RMS is 1, it implies that each Vi is around 1. So if you normalize the vector to have unit RMS norm, you're normalizing all the coordinates to be around 1, which is kind of nice for neural networks. It means that the feature vectors or the activation vectors are all well-behaved.

So we go ahead and then we just induce the operator norm kind of like we described. So on a matrix n, we define the RMS to RMS operator norm to be the max over vectors inputs such that the RMS norm of the input is one. So the average input size across the coordinates of the input is 1. And then we measure the RMS norm of the output.

And so this is inducing-- using a vector space with an RMS norm, and then there's a matrix, and then there's another vector space with the RMS norm. We induce the operator norm on the matrix. And that's what we call it the RMS to RMS induced operator norm. And as an exercise, what you can show-- you can prove that this is equivalent to the spectral norm, but with a dimensional factor.

So it's just a rescaled spectral norm with a particular dimension of prefactor. And this is also on the second question of the homework. You just need to use this. You don't need to even understand necessarily. You just need to be able to normalize something in that norm.

What is the payoff? Why did we spend so long talking about different norms and how to define them? Well, what I claim is that defining that particular norm is the thing that solves the width scaling problem. So once I know about that norm, and I use it to normalize my training, I can fix this problem.

So what's the claim? The claim is to remove drift in the optimal learning rate as I vary the width of my neural network, I should do two things. I should initialize my weight matrix at layer L such that its RMS to RMS operator norm is around 1. And then I should update by a delta w at layer l such that the RMS to RMS operating norm of my update is also around 1.

Intuitively, that means it-- the fact that I normalize the input-- the initial matrix, sorry. Because I normalize my initial matrix in this norm, if I pass in features that are coordinate wise 1 or on average 1, I can only get out features that are at most coordinate wise 1. So that's why it's a good initialization. It controls RMS norms as you move through the network.

And then why is it good to normalize my updates like this? Because it ensures that the amount that the features change by from one step to the next. We call it feature learning-- or the amount that the activation vectors can change from step to step is controlled in precisely the same way. Does anyone have questions about that? I feel like I'm just-- yeah, sorry.

AUDIENCE: Oh, it seems like you're really constraining the ability of the neural network to be expressive by controlling exactly what values the feature vector can take on. So how do you do this but not lose the approximation capacity?

JEREMY BERNSTEIN: That's a great question. So to repeat the question, it seems like constraining these properties would somehow damage the expressivity of the network, and how do we stop that? In what I just presented, we're only controlling the property at initialization, and then between consecutive steps. But over many steps, things can change much more. So I think that that's probably why it doesn't harm the expressivity. Sorry, any more questions?

AUDIENCE: This is more like an empirical observation. Like, we do the same [INAUDIBLE] more stable training where they actually have similar [INAUDIBLE]?

JEREMY BERNSTEIN: So in some sense, it's an empirical observation. And actually, the second homework problem on the homework is playing with it empirically. But in another sense, there's a lot of theoretical things to support it, the simplest of which is kind of actually just like the definition of this operator norm.

If you just think really hard about this operator norm, it tells you that things have to behave in a good way in a certain sense, but only in terms of an upper-bound. It says that the features of the particular layer cannot change more than by this amount. And it's nice to know that. But it doesn't tell you the features definitely will change by this amount.

This is precisely how the internals of the-- and there is also a theory to talk a little bit about that, but it usually makes infinite width limits. So I have some references on the final slide. It's like an active topic at the moment, and there's a lot of experiments, and there are a lot of theory. But it's still a not fully resolved kind of question.

But what I was hoping, for the purposes of the lecture, is to get across that there's quite an intuitive idea of just how big should things be, and in what norm should I measure that? And I'm trying to make this claim, which is a sort of vague claim at the moment. But if you're really careful about that, then you can make the training much more stable.

AUDIENCE: Can you help provide intuition for why this code is true? So I'm having trouble bridging the gap with why not changing the feature vector a lot actually eliminates the drift [INAUDIBLE].

JEREMY BERNSTEIN: One way to put it is that as I change the dimension of my layer, which is what we're scaling when we're scaling width-- either the input dimension, the output dimension, or both-- the RMS norm is a dimension, it's-- in some sense, it's a non-dimensional norm. My system is behaving well when the norm is 1. That would mean that all my activations are around 1.

What's the alternative? It's like, the alternative is either I scale the dimension, and the size of each activation either grows with dimension or shrinks with dimension. That would be bad scaling. And if you ensured that Euclidean norms were controlled, and then you trace back what that means for individual coordinates, you would see that either the-- I'm not sure which way around, but either the individual coordinates are growing or shrinking. The nice thing about this norm is it keeps the size of the individual coordinates actually invariant to the width, or the amount that the coordinates are changing is also invariant to the width. So that's the consideration.

AUDIENCE: And for the first position, but for the second one, why are we constraining the updates given that we are going to multiply them by learning rate anyway?

JEREMY BERNSTEIN: Yeah, so the goal is the learning rate should be independent of width-- the optimal learning rate should be independent of width. So we're trying to package all the dimension dependence in the right way into the norm so that then the learning rate-- after I normalize the updates in that norm, the learning rate, I don't need to change it with width anymore. So if you don't normalize the updates in a particular norm, then the learning rate itself needs to account for the dimension dependence.

It may be clearer, because the second homework problem you implement a normalized version of gradient descent, and you compare two for-- one without an explicit normalization, one with an explicit normalization. So it's something maybe-- let's just finish the lecture, and we can talk about this more

The last one is depth scaling. And I kind of wimped out a little bit. And also it's like an open research topic, so it's OK. But remember, the problem was that if I scale the depth, the performance can get worse unless I'm kind of careful about things.

And what I wanted to do is to think about this expression. Maybe I should have erased that and asked you. But does anyone know what this-- oh, wait, you can't see what I'm pointing at-- sorry-- what this evaluates to? Does anyone recognize that? Yeah, it's e to the x. Oops.

And so what I want to make is an analogy, where this is like a compound system. It's a big product of many, many, many things. But I scale the internals of that thing in a nice way, such that even when I take the number of terms in the product to infinity, the thing doesn't blow up, and it doesn't go to 0. So that's what I call that kind of well-scaled depth limit. That's something that many of us know from just math, I guess.

And the crucial thing is that we take x-- we think of this as the input-- and we divide it by L, which is the number of terms. And then we're going to take a product of L things. And it's really nice that because I divide x by L, this thing converges to e to the x. But if I put a different factor here-- if I put L squared, I take the limit-- I would get 0. And if I put square root of L, I take the limit-- it would blow up.

And so it's really thinking about what factor you should put there so that if I take the depth, i.e, the number of terms to infinity, it remains stable. That's the same of consideration you need to have if you want to train a residual network, and you don't want the dynamics to either blow up or go to 0 as you take the depth infinity. So just to write a little suggestion about that, think of a residual network as being a-- think of a single block in a residual network as being a function which takes x and gives back x plus the block of x.

And the block could be like a little MLP, or it could be an attention layer. It could be some other neural layer. What do I want to put in front of the blocks such that if I have an infinite number of these blocks, it's still a nice function. Probably I want to divide by the number of blocks, which I'll call L. So intuitively, you can think of a residual network a bit like this is one block. And then I can think about raising it to the power L, but that really means composing it with itself L times. And you really want to be very careful about the block multiplier.

I'm happy to talk-- there is a little more to the story, so I'm happy to talk more about it if anyone wants to talk about it. And then I put this caveat here, because it's still-- it's not completely-- there's different papers saying different things. It's not completely the end of the story. Yeah.

AUDIENCE: So does this have anything to do with as your depth increases, your complexity increases given a certain width as-- yeah, as your depth increases?

JEREMY BERNSTEIN: So the question was, does this have something to do with how the capacity or the complexity increases as depth increases?

AUDIENCE: Yeah, as you increase your depth, your training loss scales up.

JEREMY BERNSTEIN: Yeah, so what I'm saying is if-- what I'm trying to convey is if you set up your residual block, and you're not really careful about it, you can see performance gets worse as you make the depth larger. But I'm trying to say, here's a way to be more careful about it, to try to stop that problem. And if you look at a transformer code base, which I think later in the class we actually do, you'll see that a standard transformer block actually doesn't put a 1 over L multiplier.

The standard thing is actually to put 1 over square root L. And the argument is that the blocks are somehow incoherent and random with respect to each other at initialization. And if I add up lots of incoherent random things, it's a bit like doing a random walk. And a random walk moves a distance like square root number of time steps in the walk, so I should divide by square root that number of things.

So that's the rationale. And it's unclear whether that's the right thing to do or whether just to divide by L is the right thing to do. But it's really a thing, which is part of a transformer code base, is what block multiplier does your residual block have? If people have more questions, I'm happy to stay a little at the end and talk about them, but I think I should just get to the end of the lecture.

So the last thing I wanted to talk about is something that I'm-- I'm working on at the moment. So again, you can be very skeptical and feel free to not believe me. But the way that we're thinking about scaling and, more generally, optimization for neural networks is the idea of wanting to build a very modular theory. So I want to-- what does that even mean?

The problem with deep learning is that there's such a zoo of architectures that people want to consider-- that how can you build an optimization theory that covers all of them? Because someone can always produce a new architecture, and you're like, I have no idea how to deal with that.

But what if we could bake the construction of the optimization theory into the process of building the architecture so that whenever you build the architecture, you also get an optimization theory? That would be kind of cool. So the idea is to build the theory with the neural network. And we have this-- to actually formalize that and think about that, we have this idea of a module.

And PyTorch already has modules. So if you know a lot about PyTorch, you can think about PyTorch module. But a module just means something with weights in a weight space, inputs in an input space, and outputs in an output space.

And notice that that's an abstraction, which can handle an individual layer in a neural network or a full neural network. They're all members of this class of things. So even the ReLU, you can think of it as having an empty weight space, so it's also a module. But also a full transformer is also a module.

And the idea is that we're going to write a library. Think about this as writing your own deep learning package, and it's going to have a module class. And then we're going to write a library of what we call atomic modules. And these are things like ReLU, linear, Conv, like convolution, embedding. These are like the basic layer types that are part of the library.

And as you may have seen on the homework potentially, you better write the forward and backward functions for these things. You better write those by hand for the basic things. You just got to do that. So the forward is the function the thing expresses, and the backward is its derivatives basically.

But what we say is we're also going to equip our module with something else, which is a norm, which is going to be a function from its weight space to the real numbers. And we're going to say that, actually, we kind of know how to do that for the basic layers. As we said, for a linear layer, the good norm is this RMS to RMS operator norm.

ReLU actually doesn't have weight, so it doesn't need a norm. And you could imagine that maybe we could come up with a good way to equip the other layer types-- the basic layer types with norms. And we do that with pen and paper and by hand. And then we're going to write these combination rules.

So we're going to think about if we've got two modules from our library, we're going to have a way of composing them to give us a new module. And so you may have thought about forward function. It's obvious how to compose them. It just means compose the forward functions-- do function composition.

The backward, we're just going to do the chain rule. If we compose functions, we know how to compute the derivative. It's like the chain rule. And then the question is, how should we combine the norms? And I'm just going to leave this as a question, but it's something we've been thinking about.

So remember that module 1 has its own norm, and module 2 has its own norm. And then we need to find a way of combining them to give us a norm on the composite or the composed module. And it comes back to that question of how do we pick our norm?

Do you remember we were talking about steepest descent? And the big question is, how do you pick a norm that's a good match for your loss function? And it seems horrendous for deep learning because you can have any architecture that someone can come to you with, and how are you supposed to give them a norm so that they can do steepest descent? But what if there was an automatic way? What if there was a way to assign norms to the basic things in the library? And then when people compose them, it automatically gives them a norm on the composed thing. It would solve the problem in a way. So that's what we're trying to do.

And I'll just quickly tell you-- because someone asked-- you asked is that theory for, or is it all just kind of heuristics? And that is, and it's a range of things. You notice that the first two references are about infinite width limits. So people love to study how things behave as the network becomes infinitely big and to try to keep it stable in that limit.

The third paper is the paper from my collaborators-- also with Phillip-- where we're not trying to do those limits. And then the fourth one is, I think, a really interesting paper as well. That's the end of the lecture, but I can stick around if anyone has questions. Thank you.