[SQUEAKING]

[RUSTLING]

[CLICKING]

JEREMY BERNSTEIN: Hi, everyone. So today, I'm actually just going to tell you about some of my research, which is about deep learning. And again, you can take everything out-- you don't have to fully trust me in this lecture. And maybe you should be skeptical.

So the idea that I want to talk about is the idea of the way that we build neural networks is kind of like a LEGO set, where we have all these little pieces, all these building blocks, and we piece them together, and we can build a transformer, for example. But we could take those pieces, and we could build something else. There's definitely this modular structure.

So if that's how we build neural networks, why don't we also build our theoretical understanding of them in exactly the same way? And arguably, if you want to have a theoretical understanding of deep learning that really applies to any neural network that you actually find in the wild, you better build the theory in the same way that you build the neural network. Otherwise, how can it possibly hope to describe actual neural networks that you find out in the wild?

So what does that mean to build the theory in the same way as you build the neural network? So the idea is if you think about an individual piece of LEGO, imagine if you had a full theoretical characterization of that single object, which is maybe reasonable because it's a simple little piece.

Then, if we think about what are the operations that we have for combining little pieces of LEGO to build more complicated structures, and the two that we're going to think about is what we call a series combination, or putting one on top of the other, or a parallel combination, which is putting them side by side.

And then we should ask, what properties of an individual LEGO piece do we actually care about, or of an individual layer in a neural network? What do we care about? So a layer in a neural network has an input. It has weights and an output. And the question that we're going to ask is how-- the thing that we're really going to care about is how sensitive is the output of the layer in this case to both changing the input-- so you think about adding a perturbation to the input-- or what if we perturb the weights? The question is, how much does the output change? And think that the change in output gets contributions both from the change in the inputs and the change in the weights and potentially the interactions of those two things.

And what I want to argue is that, first of all, no one understands this. It's a really basic question about neural networks, but no one teaches you about it. No one understands it. But it's a really important question for a variety of different reasons. But one of them is optimization because, if you think about it, training a neural network involves changing the weights and changing the weights of all the different layers. And another reason you should care about this question is things like robustness. People talk about adversarial examples, where you change the input to the network and the output changes dramatically. So it's another question of understanding the sensitivity of the network.

And so the high-level way that I want to think about this question is let's say that we can characterize that information for an individual piece of the network, can we extend this understanding to combinations? So if we put the bricks or we put the layers into a series combination, that's a little more complicated because now there's two. Each piece has its own weight space. If we combine them, we're somehow combining the weight space.

And similarly for putting them in parallel, we get an object with a combined weight space. And now there's-- you think of this as duplicating the input, sending it to both of the pieces, and then we get two outputs now. And if you think about it, somehow the series combination is somehow a bit more coupled than the parallel combination, but we've still got to think about both of these things.

And then if we can really do this, if we can really understand the properties of individual pieces and then how those properties get transformed through combinations, then we can really think of a deep learning library, something like PyTorch or Jax. You should really think of it like a LEGO set, where you've got all these layers, and each layer should have its own theoretical description. And then we should have a system of rules for if you combine layers, the new object that you get has these properties.

And then once you've got those things, you should be able to build whatever you want. You can go and take this library and build whatever fanciful creation you like. And that should be allowed. And you should have a theoretical understanding of that thing automatically. So that's how we're thinking about deep learning. And there's actually a practical payoff already in certain axes. So I just want to tell you a little bit about that because otherwise you wouldn't know whether or not you should bother listening.

So what I want to claim is that if you adopt this perspective, it provides an easy way to fix these scaling issues that people have. So we talked a bit about this in one of the earlier lectures in the class. But you have these issues where you scale the width of building blocks in the network and the whole optimization landscape kind of drifts, even though the performance is getting better, the minima in the optimization landscape is shifting to a different optimal learning rate.

And then, actually, people have recently taken some of these ideas and packaged them with some other ideas but got much faster training. So this also relates to the homework problem about the steepest descent algorithms. But this is all in the last couple of months, but this is showing a comparison of training speed.

And the baseline is llm.c, which is like Andrej Karpathy's extremely high-performance C implementation, but with some of these ideas about steepest descent algorithms based on norms and spectral descent-type things, they actually got way faster training. So this is a huge wall clock speedup in training by integrating some of these ideas. So this is just to give you a sense that actually maybe some of this stuff matters.

So far, I've tried to put across this-- oh yeah, go ahead.

AUDIENCE: In this method, you showed that it was much faster than the other one. Do they know why it's so much faster, or just they found that this is faster, but not really sure why?

JEREMY BERNSTEIN: Partly understand why. Yeah, we partly understand why. Basically, the difference between the faster curves are using this spectral descent algorithm, which uses steepest descent under the spectral norm, which was in the homework problem. And if you believe that the geometry of an individual linear layer is somehow described by the spectral norm, then that would explain why. So it's just integrating that idea. But we'll go a little more into things. But I first want to zoom out.

And so we talked about optimization theory earlier in the class, but I want to try to connect optimization theory to this idea of building neural networks like a LEGO set. So if you remember, let's say that we have our loss function, which is like this L symbol. And we've got an n-dimensional parameter space. And then the loss function maps the set of parameters to a real number. And let's say that we can Taylor expand the loss. So if you remember, there's the first-order term, and then there's the Hessian term, and then there's higher-order terms in the Taylor expansion.

And if you remember, what we said is, what if we could come up with a norm and a sharpness parameter lambda such that we can upper bound-- at least we can upper bound that second order piece in the Taylor expansion using this norm? So yeah, we were saying, what if we could do this? What would it mean? And in particular, can we find a norm and a sharpness parameter such that this is a really tight upper bound? That would also be really great because then we can actually think of using this bound as an actual description of the loss function.

Well, if we could do that, what we said and what we spent time on the homework thinking about is this thing called steepest descent optimization algorithms, using particular norms that we can think about picking an update step to minimize this, the linearization of the loss, plus the penalty term that involves the norm. And then we can think about if I pick this norm, I get this optimization algorithm. If I pick this other norm, I get this other optimization algorithm.

So how should we actually produce such a norm? What is a good norm for a neural network? Why would we favor one norm over a different norm? I think that's an important question. So that's what we're going to try to think about over the next few slides. And just remember that-- so we're trying to produce a norm and a sharpness parameter such that the second-order piece in the Taylor expansion of our loss function is upper bounded by that object in the red box.

So how are we going to think about this? Well, first of all, remember that, in deep learning, we have what's called a composite loss function. So the total loss is the error measure composed with the neural network. So that the error measure is l and the neural network is f. So it's a composite structure. And then also remember that we have something called the Gauss-Newton decomposition. So whenever you have a composite loss function where both parts of it are suitably differentiable, we can decompose the second-order term in the loss function into two pieces. So the first one involves the curvature of the neural network, and the second piece involves the curvature of the error measure. So that step one is just like, because we have a composite, we have the Gauss-Newton decomposition.

So step two is now let's suppose that we know a really good norm on the output of the network. And maybe that's reasonable because the output of the network is often something-- it's a simple thing like a vector, like either the network tends to output a vector or it outputs a real number. So we can usually come up with a good norm.

AUDIENCE: So what would it mean for a norm to be bad in this instance?

JEREMY BERNSTEIN: That's a good question. Well, at least if you just have the output of the network, and it's a vector with the kind of there's no obvious extra structure on that thing other than it's a vector, there's only a few norms you would really think of, Lp norms, basically. But if you have a more complicated object, like a full neural network, there's a whole zoo of norms that you might think of. So it's at least like maybe you give the output the infinity norm, the L2 norm, the RMS norm, the L1 norm. There's not many other obvious choices. So that's what I mean. And again, I may be wrong about these things, so be a little skeptical because this is just the way that we've started thinking about it.

So the point is that if we have a norm on the output, we can take the Gauss-Newton decomposition, and then we can actually bound the two terms. And the reason that we can do that is essentially because you can read the term in orange as being a dot product between the curvature of the model and the linearization of the error measure.

And then you can apply the Cauchy-Schwarz inequality to that term. So the norm on the derivative of the error measure is actually a dual norm corresponding to the dual norm of the norm and network outputs. And similarly, you can think about the purple term as being like a vector times a matrix times a vector, and there's an operator norm upper bound on that thing.

So basically, anyway, long story short, you can bound these terms if you have a norm on a network output. So that's step two is just observing that you can upper bound the terms in the Gauss-Newton decomposition, so no approximations, just an upper bound.

So step three is just restating where we got up to. We've upper bounded the terms in the Gauss-Newton decomposition. And then we can now actually-- remember, our original problem was finding this red box finding this upper bound on the second-order piece of the loss function.

But we've actually reduced our problem because now if we can upper bound the first term in orange with a norm using a norm on the weights, so if we can produce a norm on the weights which upper bounds that first orange term and if we can produce a norm on the weight space to upper bound this last term in purple, if we can do that, you can plug those into the upper bound on the Gauss-Newton decomposition. It will give us an upper bound on the curvature of the overall loss function. So we've kind of reduced the problem which we didn't really know how to solve now into at least a different problem, and then the question is, can we solve that other problem?

And then I've just written beneath these two expressions that the first one is asking that the network is Lipschitz smooth in its weights, and the second one is asking that the network is Lipschitz continuous in its weights. I'll just pause briefly. So it's just to say that we started out with a problem about the overall loss function. We upper bounded this thing called the Gauss-Newton decomposition. And then we related the terms in that to properties of the network.

This is supposed to be a more pictorial or somehow visual description of what's going on. So we said that we reduced to wanting to have these conditions on our neural network, so a norm on the weight space such that the network is Lipschitz smooth and Lipschitz continuous in that norm.

AUDIENCE: So if it's Lipschitz smooth and Lipschitz continuous, that guarantees existence of such alphas and deltas?

JEREMY BERNSTEIN: Yes, but--

AUDIENCE: For any norm or [INAUDIBLE]?

JEREMY BERNSTEIN: Well, you could take these actually as the definitions of Lipschitz smooth and Lipschitz continuous and once you have them in one norm, you can automatically produce them in any other norm because all the norms are equivalent but it may be a much tighter guarantee in 1 norm than another norm. So we really want to find a norm on the weight space such that these conditions hold very, very tightly.

But just to describe pictorially what's going on, so we've got a neural network, and we think about passing an input into it, and it has weights and it has an output. And what you can think of doing is actually Taylor expanding the neural network in its weights. So remember, previously, we were Taylor expanding the loss in the weights, but now we've mapped things to actually just thinking directly about the network. And these Lipschitz conditions are asking for upper bounds on the terms in the Taylor expansion in a particular norm.

So can we produce a norm on the weight space such that we get nice, convenient upper bounds on these terms? And so forgetting everything that we talked about, about optimization, just directly looking at this slide and just directly thinking about it, wouldn't it be great if we had a norm on the weights that could predict the magnitude of these terms in the Taylor expansion? If we had such a norm, it would mean that whenever you were going to change the weights of your neural network, you would have a direct way to predict how much the output of the network can change up to first and second order.

So there's just a direct argument that having such a norm would be a really useful thing to have because we can now predict what our network is going to do in response to changing the weights. And then all of the preceding argument was saying, one of the use cases for having these relationships is that we can build an optimization theory around them, but they can be useful for other reasons as well. They're useful anytime you want to be able to just predict what changing the weights of your network is going to do to the overall function.

So what we've tried to argue is that having these conditions would be useful, and it would help us build an optimization theory, but it seems like, actually, how on Earth would we produce a norm, which is a really good norm for this reason because it seems that neural networks are really complicated, so how are we going to do that? It seems really hard. Go ahead.

AUDIENCE: I don't even know if my question makes sense, but I would like to know if people have used this approach to find approximation for other types of problems, so say [INAUDIBLE] neural network or something else, you didn't know how to approximate, and then you found [INAUDIBLE] and has this worked in [INAUDIBLE].

JEREMY BERNSTEIN: Yeah, what I'll say is that a lot of classical optimization, once you start to get into it, is all about finding a distance measure for your problem. And once you've got a really good distance measure, you can start to design clever algorithms. And people have that idea in deep learning but haven't really-- it's not really borne fruit, I would say. But it is a very classical idea to do this general thing. And then there is you can look up composite optimization theory, and there's lots of papers about it. But I would say that it's not really borne fruit in the deep learning. But it's definitely that idea is very abundant, I would say.

If you think about it, that's why we have different norms. Why do we have the L1 norm and L2 norm? Because somehow, they're supposed to characterize the structure of different problems. If they didn't, we would just only have one of them. So that's like why there's a whole zoo of norms is to somehow use them to characterize different problems.

So this is where the whole LEGO thing comes back again because it's like, how on Earth are we going to come up with a norm on the weight space of our whole network? Well, what if we can break it up into stages? So if we first think about the activation spaces, again, those are more like simple just vectors in a simple network. So as we said, it's kind of easier to give those norms because there's not that many options, like L2, L infinity, L1, RMS.

Now let's think about the tensor spaces, like the matrix spaces, for example, in the network. Once we've picked norms on the activation spaces, there's actually a way to induce something called an induced operator norm. But there's a procedure for taking two norms on the input and output of a linear operator and coming up with a norm on the operator. It's called induced operator norm.

AUDIENCE: So the activation functions usually don't have weights.

JEREMY BERNSTEIN: Yes.

AUDIENCE: So, I guess, what are we norming?

JEREMY BERNSTEIN: So when I talk about activations, I just meant the feature vectors. But then if you think about-- yeah, so I just meant the feature vectors. But when it comes to including-- if you want to think of a ReLU nonlinearity as a LEGO block, it will just be a LEGO block with the weight space is just the 0 vector space, or it just has a trivial weight space.

But we really need a norm not on each individual layer separately, although that's kind of already an interesting thing to think about is what norm would we give to each individual layer, but we really want a norm on the full weight vector. So we really need a procedure for taking all of these layer-wise norms and combining them into one meganorm and on the whole neural net. So that's what we tried to figure out how to do. So there's this idea of starting with very granular norms and then combining them until we have one norm that describes the full weight space.

So part two of the talk is I'm going to try to make this more like a formal procedure. So we call it the theory of modules. And so this is another perspective on this LEGO idea. But there's something called combinator pattern in programming, which is the idea of building complex structures by taking simple things and combining them. And you think that there's an abstract type, and you can build very simple things which are members of that type, and then you combine them and you get new things which belong to that type.

So for us, the abstract type is something called a module, which has inputs weights and an output. And that can be an individual layer in a neural network, or it can be a whole neural network that both of those things would be modules. And then given two modules, we have these combination rules. So there's composition, which is like series, combination. And concatenation is the name that we gave to parallel combination.

And again, it's just like the LEGO bricks. It's like putting them on top of each other or putting them next to each other. And this is just to show you actually you can build interesting little-- you can think of them also as circuits. But if you want to add two modules, you can put them in parallel, and then you can compose them with an addition module. So it's like a kind of little circuit.

Scalar multiplication-- you can take a module and you can compose it with a special module, which just multiplies by scalar. And then you can combine these things to build a residual block. So it kind of just looks like this. So it's like some concatenations and compositions. But essentially, then, you can think about taking that little circuit and composing it with itself a bunch of times to build a residual network. And this is just to give you an idea that actually those simple kind of binary combination rules are actually enough to build all the neural networks that you want to build.

AUDIENCE: [INAUDIBLE], I guess we're adding [INAUDIBLE]. I guess we could just have modules that have two inputs then?

JEREMY BERNSTEIN: Yeah, a module having two inputs we would think of as the input as being a single tuple. So the input space can have whatever structure you want.

AUDIENCE: Just concatenate operational [INAUDIBLE]?

JEREMY BERNSTEIN: Yeah, so it just has to be a vector space, but in the abstract sense. So this is to say that we can build-- then we have this kind of categorization of different types of modules that we find interesting. So atoms-- this is the name that we give to the simple layers that you write by hand, like a linear layer, a convolution layer, an embedding layer. Bonds are just the ones with the kind of trivial weight space. So things like nonlinearities don't have weights. And a functional attention is the part of attention without the weights. And then you could compose it with some linear modules and so on if you want to build an actual attention module. And then compounds is the name that we give to anything you build by combinations.

And this is just a recap that what we really what we've said that-- the thing that we care about or the thing that we want to know about is the sensitivity information. If I change the inputs and I change the weights, how much does the output change? Can we predict how much the output changes from knowing how much we change the inputs and the weights? And can we do this in particular for any module? So let's suppose that we can do it for the basic building blocks. Can we do it for any compound module or anything built from our LEGO set is the analogy we were making.

So I worked recently with a pure mathematician, basically. So it's quite an abstract or formal way of going about solving this problem. But essentially, we've come up with these definitions. So just to remind you, a module is something with inputs, weights, and an output. But we also give it some metadata, like its forward function, a sensitivity number, which is supposed to measure the input-output sensitivity. So it's one of the Lipschitz constants. The mass is supposed to be a kind of hyperparameter which sets how important this module is compared to other modules, how much it will contribute to learning. And then we're asking that it should have a norm on the weight space.

And anything with these attributes is a valid module but because, in our library, we don't want to have really horrible functions, we want to have nice functions that people can use, and so that's why we have this second definition, which is a module being well-normed. So these are the good modules, the ones that we actually like to use and that we want to include in our library.

So for a module to be well-normed, it needs to have a notion of a norm on its input space and a norm on its output space, which corresponds to roughly saying you want to know how the inputs behave and how the outputs behave, or how you want the outputs to behave. And then we'll basically say that a module is well-normed if it's one Lipschitz in its weights and if it's Lipschitz in its inputs where the sensitivity number assigned to the module is Lipschitz constant.

So the way that we're thinking about this is like when we say that a LEGO brick has a theory associated with it, it basically means that a LEGO brick has these attributes. And the second definition says that the attributes are based on constraints with respect to each other. And if the LEGO brick satisfies these constraints, we'll put it in our library. And if it doesn't, we'll just throw it out because we don't like it.

So I want to give you some examples of what this actually looks like. So these are some-- first of all, the linear module, the most classic does a matrix vector multiplication. We'll say that it has sensitivity 1 and mass 1. And we'll give it this kind of funky norm, this rescaled spectral norm.

And then we need to ask, is this module well-normed or not? Do we want to put it into our library or not? And so the claim is that if you set the input and output norm to be the RMS norm, which is kind of a natural choice for feature vectors inside a neural network, then the linear module is well-normed if it has bounded inputs and if the weights are bounded.

So yeah, I'm still kind of thinking about this topic, but it's kind of interesting that one of the basic things that, in an actual neural net, that you do is you normalize all the feature vectors everywhere explicitly. People don't really know why they're doing that other than somehow, intuitively, it's going to make things more stable and stop things blowing up. But this is pointing out that if you want to have Lipschitz guarantees for a linear module, the inputs and the weights must be bounded. If they're not, it can be arbitrarily sensitive.

So a useful counterpoint is an embedding module because, at first glance, it looks like it's kind of the same thing as a linear module. You can write it as a matrix vector multiplication. But the difference is that you think about an embedding layer in a neural network is you pass in a one hot vector, and it picks out one of the columns of the matrix, and that's the embedding that corresponds to that token. So it has an embedding layer has a very different semantics to a linear layer.

And one of the ways that gets expressed is by giving the weight space of an embedding module a different norm to a linear module. And similarly, when we ask if the module is well-normed or not, we'll put a different norm on the input space. And the way we think about this is that the input vector to an embedding layer is a one hot vector, which is a bit like a probability vector. So we give it the L1 norm. And then it's basically with that choice of input norm and output norm. It's well-known that under the same conditions of like unit inputs and unit weights.

AUDIENCE: Actually, I just finished trying to figure out the well-normed module definition before-- there's a diamond, like a rotated square. What does that mean?

JEREMY BERNSTEIN: That means contract over any tensor indices which are shared between the-- it's just like a tensor contraction. Does that make sense? It's like if there are two vectors, it's a dot product. If it's a matrix and a vector, it's a matrix vector multiplication. If it's a matrix and a matrix where the indices are the same, it's like sum over the [INAUDIBLE] indices. It's just like the objects in a neural network have a whole zoo of different shapes. And it's hard to come up with a notation to express that. So it's just like a shorthand for figure out what kind of multiplication you actually want to do here.

AUDIENCE: Do you not run into issues when you optimize this with boundary value problems, where, let's say, you're at the boundary of your input space being less than or equal to 1 and your gradient is like trying to push you outside of the boundary?

JEREMY BERNSTEIN: Yeah, that's a great question. So that the short answer is we've not actually implemented this part really faithfully yet. So when we actually implement it at the moment, we just let everything go outside the boundary. But I want to do that. And the idea is to design like manifold optimization algorithms where the weights stay on this like well-normed manifold. And then, because of that, that gives you guarantees about how the outputs behave. And then you can think about those as the inputs to the next layer.

And there's a way to make everything consistent. We just haven't done it yet. So yeah, so far, it's different aspects of the whole theory seem to be useful. But there's not like a complete version, which is where everything is consistent in practice.

So the reason for setting up those abstract definitions of what is a module and what is a well-normed module is that we want to have combination rules which somehow automatically preserve those properties. So if you take two well-normed modules in your library and compose them, the new thing that you get is automatically satisfies the definition of being well normed.

And so the types of things that we want to show are that the combination rules are associative. So if you do compose a chain of things, you get the same fundamental object, whichever order you take those compositions, for example. We want the space of modules to be kind of closed under combinations, the space of well-normed modules to be closed under combinations. And then we want that mass parameter that one of those attributes to allow us to tune how much different modules contribute relative to each other. And intuitively think about that as giving you the freedom to tune learning rates at different layers if you want to do that.

And so we came up with these rules. So again, these are just definitions. And the kind of style of what this is, is come up with really, really clever definitions. And then everything else is easy. So the definitions are kind of the hard part. So for composing modules or putting modules in series, you compose the forward functions, which seems kind of natural. The sensitivity, which is supposed to be the input to output Lipschitz constant, those multiply, which is kind of like if you compose things, you multiply that Lipschitz constant. So that's maybe intuitive. Mass is like, you have to think about it a little more.

And then the important thing is that we use a max norm. So when we compose modules, we want to have a norm on the joint weight space. Then we pick the max norm with a special weighting. So there's coefficients p and q, which weight the two different norms in the max. And p and q depend on the mass ratios, which that's the idea that the mass allows you to tune the importance of the two different modules in the composition, and the scalar p also couples to the sensitivity of the second module.

So that's like, if you compose two things, but the second one is very, very sensitive to its inputs. Somehow, the first one should know about that because if you're going to change the weights of the first one, you hopefully want to know that the second one is really, really sensitive or it's very insensitive. So it's like just another view on really trying to characterize the sensitivity of how things behave when you combine them with each other.

AUDIENCE: So we explained the intuition for sensitivity in terms of [? functions. ?] What's the main intuition for mass?

JEREMY BERNSTEIN: Yeah, the intuition for mass is if you just forget about all of this and you say you're going to train a transformer and you think about the structure of a transformer, it's got like the kind of body of the network is like residual blocks. There's an embedding layer at the beginning, and there's a linear layer at the output. And then you say like, OK, now I want to scale the number of blocks. And let's say you want to even scale the number of blocks to the point where there's millions of blocks, but there's only one embedding layer and one output layer.

The claim is that it's not obvious whether in the limit of a really large number of blocks. Should you even still be training the input layer and the output layer? Should they now become negligible because you want all the learning to happen in the middle? Or actually, the embedding layer plays a really interesting and different functional role to the blocks, so even if you have a lot of blocks, you still want to train the embeddings. It's not obvious what the answer to that question is. I find it difficult to know. So that's why there's a degree of freedom, which is letting you tune how much you want the embedding to be important relative to the blocks.

AUDIENCE: So it's being defined as the thing that you add up, regardless of whether it's composition or concatenation. So in that case, could you say if you had a set of blocks, and you're trying to analyze it, and you doubled all of their masses, would that give you the same analysis?

JEREMY BERNSTEIN: Yeah, it would. Analysis goes through. But it would just mean that when you train the network under this norm, using this norm to help you train it, that the amount of contribution towards learning of the blocks would double compared to the things whose mass you don't double.

AUDIENCE: Oh, but if there's no other thing that you don't double--

JEREMY BERNSTEIN: Then nothing changes. Exactly.

AUDIENCE: Follow-up question on this. How do you determine the mass for different blocks? So I am assuming here that mass incorporates some-- defines the richness of a certain block in terms of its feature representation or ability?

JEREMY BERNSTEIN: What it determines is how much that block during training is going to influence the features of anything that it contributes to, any kind of super module that it contributes to. And sort of like a degree of freedom that we couldn't really see a theoretical way to get rid of, so it's something you have to tune. But when we actually did experiments, we just tuned the mass of all of the blocks together. So we give each-- if all of the blocks have mass m and there's l blocks, each block gets mass m divided by l. So there's one number, which is the mass of all the blocks. And then we fix the input layer to have mass 1 and the output layer to have mass 1.

AUDIENCE: So if you have a block that's linear that has mass 1, and you have a block as a convolution, that would also have the same--

JEREMY BERNSTEIN: Yes. And we do that for simplicity because it's a pain if you-- but the current understanding is maybe it would work better if you tune them separately. But you can tune them in groups if you want to. So concatenation is kind of analogous, but it doesn't have the coupling because they don't compose with each other. So it's a little bit simpler. But the point of the point of these definitions is that you take these definitions. You take the definition of well-normedness, and you can prove that if you compose two things that are well-normed, then the composition is also well-normed. So they preserve that property.

And they have another-- there's another advantage which is that you can work this theory through to second order. So you define what you mean by the sharpness. It's a bit like the reciprocal of the smoothness. But you say that the module is alpha, beta, gamma sharp if it's second derivatives. In other words, it's Lipschitz smooth if the module is Lipschitz smooth with respect to its weight norm and input norm. But because a module has two arguments, we can give it three sharpness coefficients, which correspond to varying just the weights, varying the weights and the input, or varying just the input. So that's why there's three sharpness coefficients.

And you can think of these as being like generalized top eigenvalues. So oftentimes, when people talk about the sharpness of the Hessian in deep learning, what they're meaning is how big is the largest eigenvalue of the Hessian. But somehow, that's like a Euclidean constant. And if you don't believe that the geometry of the weight space is a Euclidean geometry, then you shouldn't necessarily even care about the top eigenvalue. And you need some kind of generalized notion.

And the point is that if a module is well-normed and has these sharpness coefficients, there's an automatic way to determine what the sharpness coefficients are of compositions and concatenations. And then you can use that to show that the overall loss function is Lipschitz smooth and some other things.

So that was a lot of building a theory and not really clear if it's going to be useful. So the next two parts are talking about different ways that we've been trying to apply this theory. And in some sense, the theory is intended to try to solve these problems as well.

So now I want to think about scaling. So you want to scale your training. You want to make your transformer as big as possible. And that's this idea, even in the natural world, that big things with bigger brains are supposed to be more intelligent. And the African elephant is, according to Wikipedia, is the organism with the most neurons.

And then we've got this recipe for artificial general intelligence, where it's like, get the biggest supercomputer that you can and scrape all the data and then train the really big transformer. And that's this problem that we talked about it in the earlier lecture, but the training properties of the system can drift as you scale up. So even though with the case of width, even as the network gets wider, the performance is getting better, but the optimal step size, which is on the x-axis, is just drifting. And sometimes, if you scale badly in depth, the performance can get worse as you make the network deeper. So we want to try to have a way that solves some of these problems.

So our thesis for good scaling-- why is scaling bad? Our thesis is scaling well is actually not difficult. It's just about really understanding the Lipschitz properties of the system, which is the thing that people don't really understand.

And so the thesis is that if you have Lipschitz guarantees for your network which are tight, meaning that it's not like a really loose band that's kind of useless, and if the Lipschitz constants are non-dimensional, which basically means that as you change the scale of the system, the Lipschitz constant is staying fixed, it's not varying, then if you normalize the updates in the corresponding norm, that will give you good scaling. You kind of call it a thesis because it's not really a theorem. It's just like what we think.

So we want to have Lipschitz constants for the overall neural network, which are independent of the scaling dimensions. And what we really want is to believe that the bounds that these Lipschitz bounds are going to be tight, and the amount of tightness that the bounds have is not dependent on scale.

So then, if we can have that, then the norm on the overall module, which those bounds hold in. If we control that norm of the weight updates, we'll be able to predict how much the outputs change. So that's one of the reasons we were setting this thing up. So in other words, this is the example that we mainly think about, but if the neural network is 1 Lipschitz, meaning that the Lipschitz constant is 1, so one is a really great number, which is an example of a number, which is nondimensional, then if we control the weight norm of our weight updates, we'll be able to predict that output norm of the change in outputs.

And this is just showing you some of the papers where we wrote about some of this stuff. But we broke up the problem into these two pieces. What are the properties of an individual layer? And how do we keep them under composition and concatenation? And this is just saying that, actually, some of them put a little bit on the homework, this idea that this particular scaled spectral norm is a really good norm for getting Lipschitz guarantees on a linear layer. And this paper that I worked on with Philip and also with some other collaborators, which is about the idea of combining things.

And we actually built this software package, which is still a work in progress, but it's the idea that all of those rules, everything that I was showing you, are things that can be programmed. So you can actually build this library. So you think about it as building this LEGO set, where you can combine-- so this is an example of building an MLP by combining layers and nonlinearities to get-- this MLP would then be a compound module.

And then you would automatically get its forward function through taking the composition, but you also get this normalized function. So when we compose things, we automatically build a norm on the weight space. And then we can use that norm to normalize. And so this is our simple programming example. And the claim is that because we did this clever normalization, the training will now scale better without needing to retrain the learning rate. And you can basically build it on top of different programming languages. It can be put on top of PyTorch.

And this is some empirical evidence. What we're showing here is actual width sweep. So we're increasing the width of the network. And the left plot is just training nanoGPT, which is Karpathy's transformer-- sorry, go ahead.

AUDIENCE: I didn't mean to interrupt. [INAUDIBLE] question.

JEREMY BERNSTEIN: The point is that if you just take like nanoGPT, which is this GitHub repository, and you try changing the width, the optimal learning rate is going to drift. The middle plot is our own transformer implementation, which has some tweaks to the architecture to make it actually Lipschitz. And then the third plot is adding normalization on top of the optimization. And the point is, this is supposed to be saying that by adding this normalization, we kind fix the drift in the learning rate.

AUDIENCE: So I guess from the perspective of a gradient that has not been normalized, when you apply normalization, it's as if you are changing the learning rate.

JEREMY BERNSTEIN: Yes. Yeah, that's the one perspective.

AUDIENCE: I guess it would be like changing learning rate, but in a way that's functionally dependent on the gradient itself.

JEREMY BERNSTEIN: Yes. And it changes the learning rate in a way which depends on what the architecture of the network is. So depending on-- different layers will get gradients rescaled by different amounts.

AUDIENCE: So learning rate here is a scaling of that entire process.

JEREMY BERNSTEIN: Yeah. And this is to say that things also work in scaling depth. So the baseline nanoGPT performance-- it kind of is not so bad if you just scale depth. It works reasonably well, but it's a little bit messier. And this is just to say that-- so in the middle plot, which is without applying our normalization procedure, Adam coincidentally does pretty well in terms of depth scaling, and then adding the normalization on top, things stay pretty much the same, actually.

So this is good because this is pretty much what we built this whole thing to be able to do. So it's kind of we knew these things were going to work because we had an intuitive understanding of how to do scaling, and then we kind of built the theory in order to do this. So it's like, kind of knew this was going to work. That's just saying we do some other things in the paper.

We spent a whole lot of effort to try to build this theoretical framework. And the idea that we have now is actually maybe it's going to be useful for other things. And something I'm really excited about now is this idea of duality and what we call modular duality. So what does that mean? And this is the thing which led to the big training speed-up. So that's why we care about this, or one of the reasons.

So if you remember that we suppose now that we came up with a norm such that we could upper bound the error in the linearization of our objective function. And then we remember that because we did this on the homework, that you can then derive steepest descent optimization algorithms by solving the argmin of this thing, finding the minimizer of this thing. And then you remember, because you did it on the homework, that you can decouple the solutions for that problem into two pieces. One is evaluating a dual norm, and the other is evaluating at argmax over a set of unit vectors. And so we think about the purple box as roughly being the step size.

And we can call this the second thing-- we can actually call it a duality map. And what does that mean? Basically, there's this idea in more classical optimization theory that the gradient is a dual vector and the weights are a regular vector. And the only thing that you're allowed to do with dual vectors and regular vectors is combine them. Well, basically, the dual vector is a linear map that takes a regular vector and returns a real number.

And that's borne out here by saying that if you have a gradient, the thing a gradient is really, really what it's really good for is taking dot products with weights and describing the linearization of the function. That's the only thing that you should ever really trust the gradient will tell you is the linearization of your function. The idea that you can take a gradient and just do gradient descent with it is generically a bad idea and something that we should all forget. So that's like this idea of-- and it's the idea that a dual vector, even though it may have the same dimensions, technically you could add and subtract them because technically they have the same dimensions, it can still be a bad idea.

What we need is a kind of formal procedure. We need a procedure for converting dual vectors, like the gradient, back to the primal space such that we can add and subtract them. But we should not just directly add and subtract them. But this is an example of such a procedure. And just looking at this argmax, what you can think of it as saying is I want to take a step of unit norm which maximizes the linear improvement of my objective function. And just subtracting the gradient from the weights may not solve that problem.

AUDIENCE: Is that your beef with that because we're doing nonlinear optimization, therefore, you have no guarantee that your gradient plus the step size results in an improvement?

JEREMY BERNSTEIN: Well, for a sufficiently small step size, it will give you an improvement. It may just be really suboptimal because it may not solve-- there's a sense in which this thing is an optimal procedure just in the sense that there is an optimization problem here. There's a max. There's an argmax. So the solution the thing will solve some kind of optimality criteria. If you pick a good norm, it's like an optimality criterion that you really care about. And just moving in the gradient direction is equivalent to saying that this is the Euclidean norm. And so the optimization space in neural networks is really unlikely to have a Euclidean structure. So that's the argument.

And this is just a picture where it's suggested by some other ideas in optimization. But you think about the weight space-- the space that the gradient lives in, you think of it as being a very distorted version. So even though it may have the same dimensionality as the weight space, the things that live in the gradient space kind of maybe are stretched and distorted. So we need some procedure to undistort them before we're going to subtract them from the weights.

So this is just to say that you should never do this ever again unless you're really sure that your space is Euclidean. And that you should always think, what is this dualization procedure that I really need to think through what it is? In a sense, this should not be surprising because everyone tells you, if you're going to do deep learning, you should just use the Adam optimizer. No one ever tells you to use regular gradient descent, so there should be some idea that there is, intuitively, we all know there's something bad about regular gradient descent, we just don't quite know what it is. So this second thing is supposed to pass the type check.

And the question is, how do we come up with such a dualization procedure for a neural net? Well, if we built this, we had this recursive procedure for inducing a norm on the weight space of the full neural network. So maybe there's a corresponding procedure for recursively constructing a duality map. So that's what we wanted to propose. So it's the idea that, given individual layers, if you have a norm on the weights of an individual layer, you can solve a duality map for that layer. And that's what you did on the homework because we said that we're going to give the linear layers a spectral norm, and then we're going to solve the duality map that corresponds to the spectral norm.

And then once we've solved those layer-wise duality maps, it turns out that there is a valid recursive procedure for combining them to come up with a duality map on the full architecture, which I didn't write it down in the slides, but maybe it's something fun to think about.

AUDIENCE: So this would mean that the dualization of g always has a norm of 1. Oh, never mind.

JEREMY BERNSTEIN: Yeah, it's because we broke things up into two pieces.

AUDIENCE: Oh, so it's a projection in that space?

JEREMY BERNSTEIN: Sure. So we broke up solving this argument. We broke it up into solving an argmax over unit vectors and then multiplying by the dual norm. So there's also a recursive procedure for computing the dual norm. I just didn't talk about that on the slide.

AUDIENCE: So I guess I'm interpreting it as we're finding whatever coordinate the projection of the gradient maximizes-- I guess I'm just looking at argmax of that in the next slide. We're looking at the argmax of the inner product, interpreting that as some of projection.

JEREMY BERNSTEIN: It doesn't necessarily have to be a projection. Actually, maybe it is, in some sense-- yes, maybe it is in the projection. Definitely the one on the homework was a projection. Yeah, I think, for norms, it probably usually is a projection. So yeah, I need to just think through that. But I think you're probably right that you can think of it as taking the gradient and projecting it to the unit norm ball but also distorting the direction.

But yeah, this is supposed to be you think about it as you've got some Lipschitz guarantee and some Lipschitz smoothness guarantee, which tell you how much the neural network output is going to respond to changing its weights.

And so that, in particular, one of the things that tells you is when the second-order changes and the network output become important relative to the first order changes, so a sensible thing to do is to take an optimization step which squeezes as much kind of juice out of the linearization as possible while making sure that the second-order network effects don't dominate the first-order effects, because the gradient really only tells you about the first-order effects. So that's an intuitive description.

And if you can solve that problem for individual layers, how do you solve it for a full network? And I'm just claiming that there's a recursive procedure for doing that.

And this is just to say, to tell you a little backstory, at least, why I became interested in this, is because there's an optimization algorithm that people at Google proposed called Shampoo, which won this speed challenge at this industry benchmarking competition. But if you read the paper that proposes this method this, first of all, the method that they actually use is totally different to the method in the paper. And it kind of feels like maybe there's not, let's say, a complete understanding of what the method is doing.

So I just wanted to look at this method and strip away some of the things which people don't actually use. And it looks like the core primitive in this algorithm is basically, for linear layers, it does this funny gradient transformation. And then realizing that this funny gradient transformation is actually equivalent to that homework problem that you all did, but when you read the paper, they don't frame it that way. So it's kind of interesting. But it seems to really speed up the training. So basically, what the claim is that hiding inside of this optimization algorithm that people at Google are really excited about is this very simple primitive.

And because we were working on this modular norm stuff, it's very easy to implement this type of algorithm because we just override the normalize function of a linear module to do this spectral operation. And then I implemented it in a simple notebook, just as a proof of concept, and could actually see that doing this special normalization, it actually leads to training much faster than the thing in the middle, which is just like a naive form of spectral normalization. And both of them are much better than just vanilla gradient descent.

And then there's this open-source neural network speedrunning community. And this person, Keller Jordan, made a really, really fast implementation of this spectral dualization procedure. It turns out, at the moment, it's the fastest way to train nanoGPT on this speedrun. So it's seems that's actually-- the worry with the technique is maybe, in principle, it's the right thing to be doing. But it's so expensive to do these spectral normalization operations that it's practically useless. But it turns out there's actually a way to implement this dualization procedure really fast using an iterative method.

So this is to say that what I told you-- when we wrote this modular package, we actually kind of did it wrong because when we said that we should normalize, that is actually not optimal. We should actually do this dualization procedure. And it should recursively compute the dualization based on the architecture of the network. So we still need to update the package to do that.

So in conclusion, yeah, I really like this idea that we should think about building a neural network as building LEGO, and that you should be able to build whatever you want, and you should be able to have a theoretical understanding of what you build, and it should be based on thinking about just how you combine individual bricks.

And so far, we've been thinking about these scaling issues and now, apparently, also these training speed records. And I think that this idea has not really proliferated at all really. But I think it can be a more generally useful idea. So if you want to save energy and money and time, a good way to do that is to make your models really low precision. there's no one has a theoretical understanding of how precision in a neural network should behave. So as far as I can tell, all the research is just trying stuff.

But intuitively, precision and continuity are very closely, very intimately related concepts because if you want to quantize as much as you can, you need to know what discretization scale the space is kind of continuous on or something like that. And similarly, these classic adversarial robustness things, I think, should also be related to having a good understanding of the continuity properties of the space. So you want to cast these as questions of module sensitivity.

And I want to turn this into an open-source project. So I made this website, but it doesn't have very much information there yet. But anyway, that's it. So yeah, thanks.