[SQUEAKING]

[RUSTLING]

[CLICKING]

SARA BEERY: OK, so welcome to lecture two of deep learning. Today, we're going to talk about how you actually train a neural network.

And I imagine that some of this might be review for some of you but, hopefully, the perspective or the way we talk about the process of training the neural network might be a slightly reformulated from something you've seen before. I definitely think that this slide is a nice way to characterize the process.

So today, we're going to talk about how you actually train a neural network. First, we're going to start doing a bit of a review of gradient descent SGD, high-level, some optimization ideas. Then, we're going to introduce the concept of a computational graph. We'll talk about backpropagation through chains and through multi-layer perceptrons.

We'll talk about backpropagation through DAGs. And then we will get into the concept of-- sorry-- differential programming. OK, so just, again, as a bit of a refresher, when we talk about neural network, it's usually something that follows the rough structure of this, where you will have some input data x.

And during training time, you'll have some ground truth for that input data. This is the thing that you want the model to learn to predict. So in this particular instance, that is a clownfish. You're asking this model to predict, maybe, a visual category, and that's y.

And then we have some model. This is a chain of different blocks or layers of a neural network. Each of those layers will have parameters associated with them. And then once you send this input through all of those layers, you'll calculate a loss based on the difference between what the model predicts and that ground truth.

And all of those parameters that are going through the network, those are actually what's getting learned. And what when you say getting learned, essentially, we just mean we wiggle the values around in numerical space until we get something that is at least reasonably close to optimal, based on a set of data that we actually have access to.

Cool. So essentially what you're trying to find is some optimal set of theta that minimizes that loss function over all of your available data set.

Now, when we talk about gradient descent, here, finding that optimal set of parameters, what we often will talk about is that that's going to be the set of parameters that is optimal for some cost. In this particular sense, that cost function is the sum of the loss over all the data points in our training data set.

And then how do we actually fiddle those weights around? What's the mechanism? That's really just using optimization to try to find those optimal parameters for that cost function.

So what's the knowledge that we have about J? Well, first, we know that we can actually evaluate J of theta for any given theta. So this is something that we can actually evaluate, right? You can calculate the loss over a training data set or a test data set.

We also know that we can evaluate the gradient of theta. And this is first order optimization. You might take some approximation of the gradient based on a specific point. And you can evaluate that as well. And, additionally, we can also do something like evaluating second order optimization, where you could actually calculate the Hessian.

In practice, this isn't something that we often actually do. Usually, we really just focus on that first and then that first order optimization, where we're just looking at that linear approximation to the gradient at a given point.

And then what is gradient descent? So if you're trying to find this theta star-- that's this optimal value-- what you're going to do is you're going to start somewhere on what we call the loss landscape or the surface that represents that cost function. And then you're going to move in the direction of the gradient, the maximal gradient, and takes steps over your data.

And so for each of these, you'll start moving in that downward direction on that loss landscape and eventually find a minimum. Now of course, you might note, if this loss landscape is non-convex, if it's quite complex and maybe has multiple minima, this may not be guaranteed to be the global minimum. It might just be a local minimum. And how bad that local minimum is relative to the global minimum, often, we don't know.

So if you assume you're taking a single iteration of gradient descent and trying to find better parameters, what we'll do is we will start with our k-th step of those parameters theta, and then we will define some learning rate, in this case eta, and that's going to say how far we're going to step in the direction of that gradient. And then you calculate, essentially, what is the direction of change that will optimize that loss or that cost function.

So now what you do is you take the parameters of your model, and you move them in the direction of the maximal slope of your gradient with some of learning rate step value. And that gives you the update to your parameters.

Now, this might be something that you could reasonably calculate over all of your existing training data. You'd like to, ideally, minimize the overall loss function, which would be some sum of the individual losses over every single example in your training data. But realistically, this is often not possible to calculate, mostly just due to the time, the computational complexity, especially in this day and age when data sets are getting larger and larger.

So when I started in machine learning, we were lucky if we had ImageNet scale data sets of a million images. Now we're dealing with image data sets in the billions and text data sets even larger than that.

So instead, in stochastic gradient descent, you assume that taking the gradient over some subset of your data is a reasonable approximation to taking the gradient over the entire data set. So here what you do is, instead, you compute a gradient on a subset of the data at any given point. And we call this a batch.

So if your batch size is 1, what that would mean is you would actually update your parameters after seeing every single example in the data set. You can imagine that also might be suboptimal in terms of the computational complexity of doing those updates.

If your batch size is N, where N is the full set of data, then this is just standard gradient descent. But of course, it might be possible. This is an easy to fit into the memory of your GPU, or might just be really slow, computationally intractable.

So also, if your gradient direction is kind of noisy relative to averaging over all of the examples, which is standard gradient descent, this is pretty normal. So taking a batch, you're not necessarily guaranteed that gradient over the batch is going to match the gradient over everything.

And actually in a lot of the data that I work with, where we have things like significant imbalance over the types of categories we're interested in predicting, this sampling can have a pretty strong effect in terms of how stable that gradient is. Because potentially you have some categories that are not seen for multiple batches at a time, and then you see them.

And at that point, it actually massively shifts your gradient. So this instability is actually often related to how uniformly distributed your data set is. But there are advantages of doing that stochastic gradient descent. First, it's faster. It's been shown to approximate the total gradient, even with small samples.

And that noise, it's actually an implicit regularizer. So one thing that can be a benefit of stochastic gradient descent is it can actually bounce you out of local minima and help you reasonably find closer to what might be a global minimum.

There are disadvantages. Like we said, there can be high variance. There can be unstable updates. And I've found experimentally that these updates are more unstable the more unbalanced your training data set is over the set of categories that you're interested to predict on.

Cool. Any questions about SGD? How many of you have seen SGD before? OK, that's what I thought. So this is, hopefully, a bit of a refresher for most of you.

So we also want to introduce the concept of momentum. And this is something that we might implement within our loss or within our gradient descent. So, essentially, the idea of momentum is the physical idea of momentum.

If you put something heavy on a slope, it will gain speed as it rolls down the slope. And what we do when we think about momentum in gradient descent is we, essentially, are biasing our gradient steps to continue in the direction of the previous update. You're giving it that momentum based on its past direction of movement.

And so, here, what that looks like is you're essentially just adding in, again, this parameterized by alpha momentum term that is capturing the direction that you moved in the past. And this can actually help or hurt. And the strength of that momentum, that alpha, is a hyperparameter.

A popular example of this type of momentum in gradient descent is Adam, if any of you have heard of the Adam optimizer. So this is something that is pretty commonly used.

But how this might actually help or hurt-- so here, this is just looking at optimizing for something that is quite sharp. And you can see that if the momentum is 0, you'll kind of roll steadily towards that maximum. If you actually have a momentum of 0.5, you're getting to that minimum much quicker, maybe about half the time the number of training steps.

But if your momentum is too high, you start bouncing past, and then you have this almost oscillation and wiggling that happens as you're trying to find that minimum. And it ends up taking longer to optimize your function.

And there's a really nice blog post on momentum that has some really fun interactive capacity. If you guys are interested, the link is down here. It's from back in 2017. But I found this really helpful for building intuition.

Cool. So talking about loss landscapes, which of these are differentiable? So raise of hands. Give me one that's differentiable. Yes?

AUDIENCE: The first one.

SARA BEERY: The first one. Yeah, the top--

AUDIENCE: Left.

SARA BEERY: --left, yes. Thank you. Right and left-- still hard for me. Yeah.

AUDIENCE: The third one?

SARA BEERY: Yeah. The third one, exactly it's flat, but it does have a derivative. So, yeah, exactly. These two are differentiable. So now, which of these have defined gradients in PyTorch.

AUDIENCE: All of them.

SARA BEERY: Raise your hand. Yes?

AUDIENCE: Probably all of them, right?

SARA BEERY: Yes. All of them have defined gradients in PyTorch. That's because PyTorch has autograd. It has this nice mechanism that helps us calculate gradients for any function. Which of them would be hard to optimize over? Yeah?

AUDIENCE: Second one.

SARA BEERY: Why?

AUDIENCE: It's got multiple local minima.

SARA BEERY: Yep, multiple local minima. So you might end up, depending on where you start, not in the global minimum. Yeah?

AUDIENCE: The third and the fourth, since they're both flat.

SARA BEERY: The third and the fourth, since they're both flat. Yeah, so the third one is just flat everywhere. Essentially, there's no signal in terms of the gradient. The fourth one, it's flat everywhere except for discontinuity, but anywhere on the actual graph. Again, you have no signal. Yeah?

AUDIENCE: The fifth one as well because the learning rate should be very, very small, otherwise we're just going to diverge.

SARA BEERY: So this one in the center of the bottom, this is also difficult because the gradient is very steep. It's very, very high as you get close to the minimum. And so the challenge there is that as you get close to that minimum, the gradient is really large, so then it's going to bounce you somewhere really far away. So essentially, it's really difficult to actually ever hit the minimum because of what we call exploding gradients, yeah, so definitely.

And one nice intuition around easy versus hard to optimize, you can think about flowing water. So in this example over here, you would actually, no matter where you started, you would actually end up getting to that minimum because, essentially, you can think of water flowing. No matter where it came from, it would still get to that middle.

Of course, that really only helps you thinking about, is it minimizing? So that really is only filtering out this one that has multiple minima.

So one thing that's true is that most existing popular neural network components will exhibit certain properties. So like we said, these would be hard to optimize. But now I want to go through what these different landscapes would look like, the pros and cons of them. And then we'll get into some high-level intuition and talk about where the field seems to be going in terms of the types of loss that we're calculating.

So something like this, this is like maybe the simplest case. This is a case that if you're a theoretician is really great because you can assume convexity. You have a single minimum. All the gradients point to it everywhere. And the gradient will gracefully go to 0 as the minimum is approached.

Here, we have discontinuity, but it's well defined in terms of the derivatives on both sides of that discontinuity. And this is not a problem at all for PyTorch. Here, even if this is not completely flat, which you can see here it isn't, it's what we call a vanishing gradient. The progress would be really slow, and noise in your batches, for example, might end up dominating over the signal. So this would be very difficult to optimize.

Here, this one, just zero gradient everywhere, basically, completely uninformative as how to make progress, and so it's really difficult to ever hit any low loss region. There was a question? Yeah?

AUDIENCE: Can you take a second to explain the graphs that are on the right side.

SARA BEERY: Oh, yeah, sorry. Sometimes I assume something is intuitive, and it's not. So thank you so much for asking.

So this is for different parameters. All of these are single versions of parameters. So this is now looking at that plot. Now here, this is some optimizer steps going from 0 to 100, so assuming you're starting from the top.

And it's basically stepping through, assuming this was your loss landscape, and you had some of standard learning rate, where the model would actually move throughout gradient descent, stochastic gradient descent. So in this case, there's no signal. It just doesn't go anywhere.

Whereas in the previous slide or this one, for example, you might start out over here. And it might bounce across and then go back and forth, but It will actually, eventually, find that minimum. Thanks for the question. It was a good one.

Yeah, so zero gradient-- this is what we were talking about in terms of instability. This one has what we call an exploding gradient. The gradient goes to infinity as the minimizer is approached, so you have really unstable updates, and you can really overshoot. It can be very difficult to ever actually hit that minimum.

And then, here, when we're talking about multiple local minima, where you initialize matters a lot. One way that people handle the fact that many of our lost landscapes are not guaranteed to be convex. We're not guaranteed that they only have a single local minima. This is why you might, for example, try experimentally a bunch of different random seeds, and try to then select a model that ends up looking better.

And the variance in your performance, based on different random seeds, will tell you something about how unstable your loss landscape is. So there's a couple things that we wanted to bring up. Yeah?

AUDIENCE: Question about-- you said that PyTorch [INAUDIBLE].

SARA BEERY: Well, PyTorch can make anything differentiable. That doesn't mean it will be easy to optimize. So for example, that one that's flat, it's differentiable in PyTorch, but that doesn't actually mean that it's going to find a local minimum because it doesn't exist.

So it's computationally possible to calculate a gradient there, but the gradient will still be 0. So essentially, PyTorch and autograd, they mean that even if you take something that would traditionally be non-differentiable, you can still calculate derivatives in PyTorch.

So we assume things like directional derivatives around discontinuities, but that doesn't actually mean that every loss landscape is equal. That's kind of what we're getting into here, is these ideas around what makes a good versus a bad loss landscape. Does that make sense? Cool.

OK. So one thing that we wanted to bring up was this idea of an evolution strategy. So these are gradient like. They find locally loss-minimizing directions in parameter space. And the way that they do that is they sample small perturbations around your existing parameter values and then move towards the perturbations that achieve lower loss.

So for example, you could take some perturbation or error that's normally distributed. And then when you're actually calculating your values, you calculate them of the loss function plus some amount of perturbation in that normal space.

And then when you're doing your updates, now you have this almost robustness that comes in. And it will actually select successfully minimize, even something that looks like this. So here, even if we started up here in this non-minimal situation, because we're perturbing the values of the function, it might bounce us over into a better value, just by sampling this cost function, these different things.

So this is something, these evolution strategies is something that sometimes people will take into account. And then another important one is this idea of gradient clipping. And this is a mechanism that helps us with, basically, peakiness or pointiness in our loss function. So we'll successfully minimize this function.

And so, here, it's quite simple. If the gradient gets too high above some large value, you just scale them back to that value. So you essentially just put a clipping filter on top of your gradient. And this is a useful and commonly used hack.

So here, essentially, what you do is you just calculate the change in your parameters. And then you move in the direction of that maximal change but clipped to some maximum value. So you won't let your model move too far in any direction. And this, then, helps us with not oscillating back and forth across something.

So what then is important in a loss function? So a few things that are consistently showing up-- one-- though this is, of course, not required-- everywhere continuous, another one, everywhere differentiable. And the third one would be everywhere smooth.

So one of the most commonly used non-linearities is ReLU. And this is something that is everywhere continuous, almost everywhere differentiable, but it's not everywhere smooth. We have this kink in it.

Now, recently, there's been a lot of work that will instead use something like a ReLU, which is a Gaussian error linear unit. And this one, essentially, just takes the parameters times phi of the parameters, where this is the cumulative distribution function for the Gaussian distribution. It looks something like this in two dimensions. And this one is everywhere continuous, everywhere differentiable, and everywhere smooth.

The important thing I want to point out here is that we can agree that these three things might make optimization easier. And even if we don't precisely know experimentally or theoretically, which properties are needed for training neural networks, trends do seem to be moving towards functions which satisfy all three, so things like this GeLU. Any questions about optimization and SGD? Yeah?

AUDIENCE: What was the [INAUDIBLE], use that for again?

SARA BEERY: So evolution strategies?

AUDIENCE: Yeah.

SARA BEERY: Essentially, instead of always moving in the direction of the gradient, you just randomly sample around your current parameter values, and then you move in the random direction that maximizes or that minimizes your cost.

AUDIENCE: [INAUDIBLE]

SARA BEERY: Essentially, it's helping you maybe handle things like vanishing gradients or some of these parts of your loss landscape that might be not actually functional to optimize super efficiently. And I think you can almost think of it-- one intuition, It's almost like regularization in your loss, if that makes sense.

AUDIENCE: Also, follow up, how common is it that people use everything together?

SARA BEERY: Yeah, I mean, I think everything's a bit of a big, soup pot these days. There's a lot of these different experimental tricks. Some of them, even if they seem quite different, almost seem like they achieve the same ends, when you talk about the experimental performance.

I would say that a lot of the really top-performing, state-of-the-art models on different types of benchmarks are often ones that incorporate a lot of these tricks, different hacks and mechanisms to try to get a little better at finding some maybe slightly better optimum, but I don't think we actually have a really great recipe for given a certain data set, given a certain architecture, what is the right way to optimize that architecture? Yeah?

AUDIENCE: Can you please explain how ReLU and GeLU relates to this question of [INAUDIBLE]?

SARA BEERY: Yeah, so, again, I'm not I'm not saying here that GeLU is much better than ReLU all the time because that's not necessarily true. It's just pointing out that the trend in these different loss terms for neural network training does at least seem to be moving away from functions that don't satisfy all three of these. But again, we don't necessarily know, or we haven't decided as a community or even gotten close to some theoretical guarantee as a community as to what is the optimal mechanism. Yeah?

AUDIENCE: Are these loss functions? Are these activation functions?

SARA BEERY: These are activation functions. Yeah, yeah.

AUDIENCE: Always positive [INAUDIBLE]?

SARA BEERY: Hmm? Sorry?

AUDIENCE: [INAUDIBLE] loss functions also not be negative?

SARA BEERY: Oh, well, it depends on the way you've set up your loss. You definitely can have loss functions that will have negative values. I mean, that depends, right?

And I guess what you're saying is, if you're trying to minimize a cost, you want the gradients to move in the direction of that, but that's not guaranteed if you can't guarantee the loss landscape is convex everywhere. There will be parts where it goes up and down. Yeah?

AUDIENCE: Is there any issue with having a non-monotonic function?

SARA BEERY: A non-monotonic activation function or loss function?

AUDIENCE: Activation function.

SARA BEERY: I mean, there definitely exist, in theory, activation functions that are non-monotonic, but I don't think that they're often widely used. I mean, it seems to make sense.

AUDIENCE: [INAUDIBLE] seems common.

SARA BEERY: It's not, right? Yeah, it has this part where it dips down right here.

AUDIENCE: Yeah, that's what I mean.

SARA BEERY: Yeah. But then this is the question. Is it better to be smooth or to be monotonic? And that's where I think the jury's a little bit still out.

But, yeah, I guess we could put monotonic here, but I feel like these three are the ones, at least from my perspective, that seem to be captured most consistently across the activation functions that people tend to use or are moving towards. Yeah?

AUDIENCE: [INAUDIBLE] for the perturbation, can you talk about it from a numerical mathematical equation? What are the strength of levers that affects that, making the magnitude bigger or smaller, and also from the application sense? If you're looking at a data set, what--

SARA BEERY: You mean this?

AUDIENCE: Yeah.

SARA BEERY: I mean, so definitely those parameters are something that you would need to experimentally find optimals for, for any given data set or problem. It's not like there's some universal ideal here. And I think it will probably depend a lot on which data set, which architecture, for example.

So I don't have any big rules of thumb for you there. Though, I would say one generally good strategy is you take a paper that used it, and you look at the values they used, and you start there, just usually because that they've already done it, right? They've already done it-- defined their optimal parameters.

And sometimes those can be really far off for a new data set, but it's often not a bad strategy to at least start with something that has trained for someone in the past. OK, I'm going to move on. Great questions though, guys. It's much more fun for a professor when people are engaged, so it's great to hear from you.

[LAUGHS]

Cool. So I'm going to move on to talking about computation graphs. So we're going to move to thinking of this all from this perspective of differential programming. Essentially, here, we're talking about computation graph. Here, this is something like a graph of different functional transformations. Each of those functional transformation is a node in the graph.

And then when you string all of those together, you can perform some useful computation. So intuitively, maybe this is some sort of decision tree that's a very simple version of a graph of functional transformations. So maybe I'm saying, all right, I'm going to try to make it to the classroom.

So I'm going to go into a building. And then maybe on the right, you've gone into building 45, which is correct. And on the left, you've gone into building 32, which is incorrect. And then there's these different decisions you might make based on the values of your input at any given time.

So if you did go into building 45, then you could maybe take the elevator or take the stairs. And that decision over what action you'd take could be a really, really simple heuristic version of a functional transformation because you're transforming your inputs to some output.

Of course, what we actually see here when we think about machine learning systems is each of these maybe blocks are something like a layer in a neural network or even an entire neural network. And there's nothing that says what the size of these are, any of restrictions when you're thinking about it really from this high-level perspective of a computational graph.

So deep learning primarily deals with DAGs. These are directed acyclic graph. So directed means the information only goes one direction on any given edge. Acyclic means you don't have loops or cycles.

And we also assume that every single one of these nodes is differentiable. Though we did just say that using PyTorch, you can calculate a derivative for almost anything, so that becomes less of a hard constraint.

So if we now think about this from the perspective of, for example, a really simple multilayer perceptron, like we talked about on Thursday. Here, you can think of that now, again, as one of these directed acyclic graph.

You have your input x. You send that through this linear layer, which is one node in the graph. Then that gives you some hidden unit z. You send that through an activation function or ReLU that's, again, some computation that takes an input and changes it to create an output.

Now, another hidden layer h, hidden state h. And then you take that and run it through another linear layer. And that gives you your output y. So even an MLP-- easy to represent as a computation graph.

So now let's talk about what a forward pass actually looks like and what's required to calculate a forward pass. So in the simplest sense, for any model, for any set of parameters, for anything within that function, you take in some input value.

You take in some theta, which are the parameters of that component or of function. And then you map through that functional transformation, the inputs, the parameters, and you get out the outputs of that layer. So that's a forward pass for any type of computational function.

Now, you can have multiple layers, right? This computation graph, for example, could represent an MLP. Now you have your inputs and your parameters for that first component. Then, you get those. That's an output. That becomes the input of the next component, et cetera, et cetera.

And then, at the end, maybe you have something like this loss that you use to calculate your cost. So then when you're thinking about learning, now here, in order to calculate this learning, that's where we think about computing the gradients of the cost with respect to all of those model parameters that happen all the way through the network.

And so by design, every single layer will be differentiable with respect to its inputs and, therefore, this is actually possible to compute. It is possible to compute that gradient with respect to all of these model parameters. So you can figure out how you need to update those model parameters to move to a more optimal cost.

OK. So a bit of an aside, just in case any of you are not brushed up on your matrix calculus because this will show up a lot. We're going to be doing a lot of matrix multiplications in this class. Hopefully, this is super review for all of you, but just a reminder.

So we're going to assume that our inputs are, for example, column vectors of size n by 1. Now if we define a function on that vector, y equals f of x. If y is a scalar, then the derivative of that output with respect to x, the input, would be a horizontal vector, where for each element of that vector, you're getting the partial derivative of y with respect to that first element, or second element, or third element of x. And then that is a row vector of size 1 by n.

If y is a vector of n by 1, then we assume the Jacobian formulation of the partial derivative of y with respect to x, where now you, essentially, end up with something that's a matrix of size m by n, so m rows, which is the size of y, by n columns is the size of x. And each element of that matrix will be the partial derivative of that element of y with that element of x,

Now if y is a scalar and x is a matrix-- so, for example, an image of size n by m-- then the partial derivative of y with respect to that matrix x will be a matrix of size m by n, where now here, each component corresponds to that relevant component of x. And note that by taking that derivative, we flipped the rows and columns. So it's been transposed.

And according to Wikipedia, the three types of derivatives that have not been considered are those involving vectors by matrices, matrices by vectors, and matrices by matrices. Notation is not widely agreed upon. We're now going to see any of in this class.

Cool, so chain rule. For the function h of x is f of g of x, how do you take the derivative? Who remembers the chain rule? Yeah?

AUDIENCE: x times the derivative of g of x plus f prime of x times g of x. You're basically [INAUDIBLE].

SARA BEERY: OK.

AUDIENCE: [INAUDIBLE]

SARA BEERY: I don't think that's right. [LAUGHS] It's all right, man. It's all right. [LAUGHS]

AUDIENCE: [INAUDIBLE]

SARA BEERY: OK. Go for it.

AUDIENCE: g x prime with the f prime of gx.

SARA BEERY: Yes. So the derivative of x f with respect to g of x times the derivative of g with respect to x, yeah. Good job, dude. Way to go for it. No one's going to judge anyone for getting something wrong in this class.

[LAUGHS, APPLAUSE]

All right, so then, actually, if we write this out as z is f of u and u is g of x, then you can write it out this way. So essentially, you can write this out as the multiplication of two partial derivatives. So that means that p would be the length of vector u. And m would be-- wait, what the fuck is p? What am I talking about?

All right Essentially, what we want to get at is what size are each of these things? So who can tell me what the shape of the partial of z with respect to x would be when x equals a? I can give you one freebie, if it's helpful.

So if we say the size of you is p, the size of z is m, and the size of x is n, then the partial of z with respect to x would be an M by N matrix. So what size would the partial of z with respect to u be? Yeah?

AUDIENCE: m by u.

SARA BEERY: m by p. This is really confusingly written. I'm going to rewrite this next year. p is the length of vector u equals the size of u is confusing. But, yes, m by p. And then what's the last one?

AUDIENCE: p by n.

SARA BEERY: p by n, exactly Cool. So therefore, if you said the size of z is 1, the size of u is 2, the size of x is 4, then essentially what you're saying is this derivative would be, essentially, equal to this product of two matrices, though one is really a vector.

All right. We're bringing this up because we will see a lot of matrices times matrices in the next bit. So that's the end of our little bit of a vector calculus or matrix calculus reminder. So let's talk about backprop.

How does any of this become possible? How do we calculate the update in those parameters? Essentially, what we need to calculate is the partial derivative of j with respect to, for example, those initial layers in the model, theta 1.

So by the chain rule, you can essentially factor it out. And you can map that back through all of these different layers, so the inputs and outputs of x as it goes through all these different layers. And if you need to calculate the partial derivative of j with respect to that second layer's parameters, you'll note that it's the same essentially.

But the thing that's different is, here, these last terms are different. But everything for those two are shared. And this is really the trick, right? We could separately compute all of the derivatives using the chain rule, but because these terms in the gray box are shared, we only need to compute them once.

So back propagation is a pretty simple algorithm for propagating shared terms through the computation graph. It's basically an efficiency trick, but it's one that makes it computationally practical for very large models to be able to actually calculate these gradients.

So if this is a forward pass, we're sending data forward through the network, and then computing the outputs and calculating the overall loss. Then a backwards pass look like this, which is that we send our error signal, the gradients, backwards through the network from the outputs, and the loss back to the inputs and the parameters. Does this make sense? Cool.

So what is a backwards pass look like for a generic layer? We're going to keep track of two kinds of arrays of partial derivatives here. So first, L is the gradient of the layer outputs with regard to the layer inputs. This will be a matrix, right? This is, what way does each of these parameters need to change to move in the maximal direction of optimization?

So here, this is essentially a matrix that's just telling you which would these parameters optimally move. And then second, we're going to keep track of g. And this is the gradient of the cost with respect to the activations. And this will be a row vector. So this is basically saying, yeah, essentially, what's the partial derivative of j with respect to x, with respect to that input?

So now, if we want to update the parameters, that's actually really easy if we have both L and g. Essentially, it's just a multiplication, right? You have g out, and you multiply that by L of theta. And that matrix multiplication actually just updates your layer. That tells you which way to move. And so now you take your parameters, and you change them by your learning rate multiplied by that update.

So how do we actually get L and g for each layer? So L comes from that derivative function of the layer, which we assume is provided. We assume for any given layer, that it is differentiable, and that we have a function that lets us take that derivative.

And then g is something that can be computed iteratively via recurrence. So the input gradient will always be equal to the output gradient times that L for that layer. So this is, essentially, the back propagation of the error signal back through the model. And then all of this machinery is used to then compute parameter update directions, so which way do we need to go?

So the full algorithm-- forward then backward. First, we're moving forward. Then we go backward. And then we update and repeat. And this is, essentially, how you train a neural network, right?

You do a forward pass, calculate loss. You take that loss, propagate it backward, update all your parameters, and do it again, and again, and again, until you found some likelihood that makes you think that your model is finished training. Often, we'll do this-- we'll talk more about mechanisms for this later, but it's often using something like a validation set and looking for some sort of plateauing of change on that validation set.

So if we're doing back propagation, now what we've got here, in the next set of slides, are essentially cheat sheets for a lot of this stuff. It's really just intended to be a pretty nice resource if you need to refer back and think about how you might need to implement some of these.

So now a forward pass through some hidden layer. So now we have the layer before and the layer after. We take this input. We run it through the function, and then we have this output.

Now when we're doing that backwards pass, we're taking the derivative of with respect to the output. We take the partial derivative with respect to the input. And then we also are calculating how we're taking the parameters in, and then we're calculating the change to those parameters going out.

So this means that layer L for this explicit layer-- so all the inputs are going in with the full lines. And the outputs are always in the dotted lines here. So the inputs are then the previous layers output, the change in the parameters with respect to the next layer, and the current parameters.

And then the three outputs of every layer are going to be the output running through the function, the change in the parameters with respect to the previous layer, and the actual update to the parameters. So given those inputs, we just need to evaluate those three things. And this actually makes training happen.

So now if we take a summary-- forward pass-- we'll compute the outputs for all the layers. Backward pass-- we compute the loss derivatives iteratively from top to bottom. And then the parameter updates-- we compute the gradients with respect to the weights. And then we update all of those weights.

So now what we actually do is we do this over batches. We're not doing this for a single data point at a time. And technically, what that means is we want to minimize the average cost over lots of data points, the entire size of a batch. And in this day and age, with very large memory GPUs, that batch size could be thousands, thousands of data points at once.

So then the gradient of the total cost is just the average of all the gradients of all the per data point costs. Because when you take a derivative, it can move inside the sum, so pretty straightforward. So now if we actually have this as a linear layer, let's talk through what that looks like. Because it turns out everything gets really nice and simple.

So first, forward propagation is quite simple. If we have a linear layer, the weights of that layer are just a matrix. So now if you want to calculate the output. You just take that matrix and multiply it by our vector of inputs, so pretty straightforward.

If you want to calculate backpropagation to that input, this is essentially the input gradient is going to be the output gradient multiplied by the update to those parameters with respect to the input. So essentially, that just looks like this, right? This is just that matrix that we've defined that represents the direction that you would want to move in for those weights.

And then we look at the component of the output with respect to the j-th component of the input, the i-th component of the output with respect to the j-th component of the input. It turns out that, that is Wij. And so that actually means that this is really just the weights again.

So this is really nice and simple, right? In the opposite direction, you multiply the gradient by the weights to get the gradient coming in. Cool, so your forward and backward pass are just multiplying by your weight matrix, just in a different order. Pretty nice.

And finally, now let's see how we use those outputs to compute the weights update equation. So how do we actually do that backprop through to the weights? So that looks like this, right? We're trying to take the partial derivative of the cost with respect to those weights. And so that's going to be the output gradient multiplied by this partial derivative.

If we look at the parameter Wij and look at how that parameter changes the cost, only the Ith component of the output is going to change. So this means that, essentially, you can break this down through this simple almost decomposition. And you end up getting this because of what we just pointed out, so essentially the partial derivative of the output at that i-th component with respect to the weights matrix.

The corresponding element ij is actually just going to be equal to the input of j. And so that means the following. Essentially, when you want to calculate the update to the weights, all you're doing is you're multiplying the input to the function by the output gradient. Again, it's a really nice simplification that comes out just because everything is linear.

So then when we actually want to update the weights, you're literally just taking eta, and transposing this, and multiplying it or adding it in to the weights. Any questions about of very nice, simple reformulation that gives us essentially a bunch of matrix multiplications? Yes?

AUDIENCE: Do we have enough time? Can you work through a small example with the computational graph on the board or something?

SARA BEERY: So at the end of the lecture, I actually do have an example of that. And so either we'll get to it at the end-- I'm a bit worried about time-- or it's in the slides. And so you can actually work through it. And there are the solutions at the end. Sounds good? Cool.

All right, so cheat sheet for a linear layer. The output is equal to the weights times the input. The gradient in is equal to the output gradient times the weights. And the update to the weights is equal to the input times the output gradient. And then you actually do that update based on adding in that, basically, direction you want to move in multiplied by your learning rate.

So now let's look at a whole multilayer perceptron. What happens if we put all these operations together? So first, you have some linear layer. Then, you'll have some ReLU, some activation function.

In particular, here, we're talking about ReLU. Then, you have another linear layer. So now this will be weights one, weights two. And then you calculate some loss, based on the difference between your output and the ground truth.

So now to do a backwards pass, you essentially assume a slight change in convention. This will clarify this nice connection between forward and backward directions. So instead of representing gradients as row vectors, we're going to transpose them and treat them as column vectors. And then that backwards operation for the transposed vectors will follow from that matrix identity, that AB transpose is B transpose A transpose, if you remember basic matrix math.

So essentially, this then reveals this interesting connection between forward and backward. So backward for a linear layer is the same operation as forward but with the weights transposed. So here, now you're going backward, you get, again, weights transposed times the gradient.

And then this is an interesting thing. A ReLU wouldn't be a ReLU on the backwards pass because you don't want to re pass them through ReLU. Essentially, what you need to turn it into is roughly like a gating matrix.

This will be parameterized by the function by the function of the activations from that forward pass. So this would be like a, b, and c, the activations that you had sending those weights through the ReLU on the forward pass.

What this is doing is making sure that you're not passing gradients for the components of the weights that corresponded to the parts that were in that 0 part of the ReLU. You don't want to send gradients for parts that should be masked out. Yeah?

AUDIENCE: Why is that matrix diagonalized?

SARA BEERY: That's just so that it's operating on each element independently. Cool. And then, now, you can mask the values of the gradients, make sure you're not sending gradients for things that are going into that 0, that negative part of the ReLU. And then, essentially, what this means is that backprop is still a linear model. Because even that ReLU, this is still a linear operation that we've taken.

And there's a cool intuition. Phil told this to me, and I really liked it. If you're thinking about a loss landscape, no matter how complex that loss landscape is, if we're taking a first order approximation of the derivative, essentially, what we're doing is we're fitting a plane to this complex curved loss landscape, and then we're moving the direction of the plane.

So by definition, it must be linear. We're moving on a planar surface, even if the actual underlying loss is not planar. Make sense? Cool.

So this means that forward and backward for a linear model is actually all just one really big, flipped out linear model. You can treat the entire thing as a single iteration, as just one big linear model. And it can be thought of as a single forward pass, in a way, even though the actual had a forward pass is only the first half, and the backward passes the second half. Yeah?

AUDIENCE: Comparing forward and backward pass, you said that to compute the backward pass, you need to have the x in at each layer. So I think that you need to have in memory for all the layers what is the value of the embedding. But as in the forward pass, you don't need it.

You only need it for the-- you can discard the previous embedding, for instance, with your [INAUDIBLE]. Once it has passed through the first layer, then you can just erase the [INAUDIBLE] from your memory as you're only going forward. Whereas in the backward pass, you need to save all of them.

SARA BEERY: Yes. So you do need to save the intermediate representation so that you can efficiently compute back propagation. That's true.

AUDIENCE: So there is not real symmetry [INAUDIBLE] the forward and backward passes.

SARA BEERY: No. But but, I mean, you can actually even see that here. There's this asymmetry in where the information is flowing. Does that make sense?

AUDIENCE: Yes.

SARA BEERY: Yeah?

AUDIENCE: Is there a specific reason why we don't see [INAUDIBLE]?

SARA BEERY: Oh, that was just for simplicity here. So, yeah, in practice, there would be bias terms floating around everywhere, but we just wanted to capture the intuition. Yeah?

AUDIENCE: When we're going back through our ReLU layer, do we need to know the pre-activations-- we need to have that in memory?

SARA BEERY: You basically need to save off what the value of the activations were for each component of your input. And you keep those in memory, and then you build that sparse matrix that lets you actually gate those going back through. Yeah. Cool.

OK. So let's move on to talking about DAGs. So what if your DAG isn't a chain? And actually, this is pretty common. We often have connections within our machine-learning models that are not just chains, for example, sharing weights, or splitting weights, or having things come together, be concatenated from different heads or being split into different objectives. So let's talk about what this actually looks like.

So there's actually only two operations that you need to make everything we just talked about work for any arbitrary directed-acyclic graph. And those are merging and branching. So assuming you have some merge where, here-- and that merge could be a lot of different things. It could be addition. It could be concatenation, what have you.

You have some components, some functional components that you're merging together to move forward through your directed-acyclic graph. Now, if you actually need to propagate a gradient back through that, essentially, you just need to make sure that you track the gradient with respect to the correct input variables.

So if, here, you just only pass the gradient that corresponds to the x superscript a, and on the other path, you only pass the gradient that corresponds to the components of that other part of the path. Similarly, if you're branching, so now you're splitting up your information in some way or even just duplicating it.

Now, if you're propagating a gradient back through that, it's as simple as a sum. You just take the gradients from both, and you sum them, and you send that back through that input. Cool.

So there's a much more detailed derivation in the lecture notes, if you're interested. But so now let's talk about what this means for parameter sharing. So say you have some chain. And now you have the parameters are separate for each component of that chain.

You might instead have something where those parameters are actually shared across those different components. Maybe you're reusing parameters that were pre-trained on ImageNet or something, and you're using them in multiple parts of your network. There, again, you can actually just think about flipping this on its side. All that is a branch, right?

So now, if you're passing your gradient back through that branch, you just need to be summing the gradients that are coming in from these different branch dimensions. So parameter sharing, some ingredients.

All right. So towards the end of the lecture, now I want to move on and talk about this kind of meta concept of what differential programming is and maybe why neural network training is, essentially, just differential programming. So when we think about deep learning, generally, we think about it this way. You have some network, again, a DAG. And then you define these forward and backward passes through that network.

Differential programming takes the same perspective, but now we basically just are actually programming that model. And this is really what all of these different existing languages-- PyTorch, TensorFlow, if anyone still uses it, JAX-- these are all basically libraries that enable us to really efficiently do differential programming.

So deep networks are popular for a few different reasons. First, they're easy to optimize. Everything is differentiable. Second, they're compositional. It's essentially like block-based programming.

And so differential programming surfaced as an emerging term for general models with those properties. These tweets are a bit old now, but I still think they're funny. Yann LeCun said, "deep learning has outlived its usefulness as a buzz phrase, deep learning, est mort. Vive differential programming." Essentially, even years ago, this was like maybe this is the new buzzword for what we're talking about here.

Tom Ditterich, who is significantly less splashy in his tweets, said that, "deep learning is essentially a new style of programming, differential programming. And the field is trying to work up what the reusable constructs in this style are. And there are some that we actually know. Some of these blocks have been well defined, things like convolution, pooling, LSTMs, GANs, VAEs, memory units, routing units, et cetera."

So this is taking that perspective. This is a paper from 2017 called Neural Module Networks, where essentially, here, you can see that there are these different components of the model. And some of them might be architectural, and some of them might have parameters that are learned or not.

So you're sending in something like, where is the dog? And then maybe that gets routed to a parser, and that parser might not be learned at all. It might just be a standard parser. So that component might not have any differential parameters that are actually being learned.

But then, potentially, there's a CNN down here that's parsing the objects that are seen in an image. And that might actually be something that you have updates being propagated to from your training data set.

And there's an interesting blog post from Andrej Karpathy that talks about this as what he calls Software 2.0. Essentially, here, he's saying that Software 1.0, which is the software we had originally, where everything is defined, that might be just that little, red dot. Software 2.0 is actually defining a space of possible software systems, and then you're actually optimizing within that space to build the system that you need to solve your problem.

So here, instead of thinking about this DAG, where every component is actually differentiable or every component is differentiable-- where every component is learned, instead, you might have a bunch of the components of your system be explicitly programmed by a human.

A really obvious example of this is we often explicitly program the normalization values that we want to use for natural images directly into the pre-processing of our data. This is something that we just define. We don't learn what those normalization values should be.

And then there are these other components where this part is actually programmed by back propagation, so, for example, programmed by tuning the behavior to match the training examples. And the cool thing is because we assume that every piece of this system is differentiable, you can actually optimize any node or any edge with regard to any scalar cost.

So here, you could think about calculating how the cost would change when the weights of the specific yellow function change. You could also think about looking at how the cost changes when the input data changes, right? You don't actually need to specifically do this for weights. Yeah?

AUDIENCE: So is this how we create a metric that can determine what might be good for a human to train versus the computer to train? Or is there another qualitative or empirical metric that we might use?

SARA BEERY: So a big question. So he was asking if this is something that helps us build metrics around what maybe we might want to optimize via data versus what we want to just define directly. I think, experimentally, we often find those things out for different applications.

What parts does it actually good to-- essentially, anytime a human is programming part of this system, it's constraining the system. And that constraint can be useful, but it could also be unuseful.

I think about when I started in machine learning, it was built around-- I'm going to date myself right now-- but we did a lot of what we called feature engineering, which is essentially like us building a bunch of explicit constraints into how the data might be processed to then go through a very, very simple neural network or even something more simple than that, something like an SDM.

And those bottlenecks, it turned out, were not necessarily optimal. We were defining this. We were programming it, but it turned out our best ideas were not as good as just a much more complex, larger models that were learned somewhat end-to-end. Does that make sense? So I think the jury's still out.

When you talk about metrics, that's really where that red square comes in. You define what that cost function is. So you have to figure out how you are going to define the loss and what that then looks like for your cost. And that definition will affect what your optimal are.

Cool. So let's talk a little bit about optimizing parameters versus optimizing inputs. So this is the standard approach, right? This is looking at how much the total cost is increased or decreased by changing the parameters. This is all the different parameters throughout the network. This is what we commonly think of, but you can also think about optimizing the input for a fixed set of parameters.

So here, this could be something how much the chameleon score is increased or decreased by changing the input pixels. And so here, now, if you think about this from a way where you're optimizing maybe class logits before the softmax or optimizing the class probabilities after the softmax, the maybe easiest way to increase the probability softmax given to a class is often to make the alternatives unlikely, rather than to make the class of interest likely.

Whereas if you optimize pre softmax logits, this tends to actually be a bit more stable. So what we find is if you want to actually make an image that maximizes the cat output neuron-- again, we are doing this maybe pre softmax-- it might look something like this. So this is an interesting, again, blog post back from 2017, but on feature visualization.

So this is telling you what a given trained model thinks is most cat like. And this is something you can learn. Or you can actually also do things that say, for example, make an image that maximizes the value of a neuron, j, on a layer, L. So using this as a mechanism to probe what the model is paying attention to. So that might look something like this.

This is saying that this particular neuron and this particular layer is looking for stuff that looks like this in a hand-wavy way. And this is the idea behind, for example, DeepDream. I don't know if any of you saw this, a couple of years old now. But I think these are psychedelic and beautiful. And this is looking at what these models, I guess, dream of.

And this is an idea of what's to come. But if you're comfortable with the idea that you can optimize anything with respect to anything, here, we're looking at maximizing clip as a model that tries to make descriptions and images that correspond to them close together in a joint embedding space.

So you want to build a text encoder and an image encoder where things that are similar, semantically, from text to images, are quite close together in the learned embedding space, that projection that you're projecting each of them into. Now, if you have something like CLIP, then you can, for example, tie your CLIP model together, the image encoder and the text encoder from your CLIP model, together with a GAN, something like a generative adversarial network.

And then you can optimize the embedding or that input parameter that goes into that image generator. And you can figure out, for a given input, how to optimize this so that your output of your image generator is the thing that's closest in CLIP space to any given prompt. And if you actually look at what that learns over time, you can get stuff that's fascinating.

So this is a way to actually generate an image from text. And here, again, all these trapezoids are neural networks. You can plug them together. You can take the components trained in one way and use them in another way. Really, the idea is you can optimize modules with respect to all the other modules, and the world's your oyster. Cool.

So what we covered today-- review of gradient descent and SGD, computation graphs, backprop through chains, backprop through MLPs, backprop through DAGs, and differential programming. Any questions? Yes?

AUDIENCE: I have a question about [INAUDIBLE]. So first, why do you have to train the network [INAUDIBLE]?

SARA BEERY: So essentially what we're doing here is you had CLIP. You have this text encoder and this image encoder. And they were already trained, and now they're fixed, right?

So the idea is, here, the text and the image encoder, actually, we're not backpropping through them at all. They're now defined. They're not being learned.

Similarly, you've separately trained an image generation model that takes in any given embedding and generates an image from that embedding. Now that one is also fixed. So now you have a lot of parameters in a model, but all of them are fixed.

And the only thing you're optimizing is this hidden parameter that's going to be the input to the image generator, such that this is as is maximal. So you're trying to find the sine of z value that maximizes this with respect to that input text.

AUDIENCE: So

SARA BEERY: Sorry?

AUDIENCE: [INAUDIBLE] optimization [INAUDIBLE]?

SARA BEERY: Yeah, so all the parameters here are fixed. But you are going to need to calculate how to backprop the signal back through these models to get to that one. Yeah?

AUDIENCE: Can you explain more about embedding [INAUDIBLE]?

SARA BEERY: Yeah, absolutely. So when I say embedding-- I might also say representation-- these are all terminology that we often will throw around in the machine-learning community. But usually it means something like this.

So you have some neural network. And we'll call them an encoder sometimes, again, just terminology. But this neural network is essentially building a mapping from a high-dimensional input or a complex input, something like an image, to a low-dimensional representation or a low-dimensional embedding of that input data.

And so that might be a vector of length 2048, for example, or 1024. Those are both common embedding sizes. But really, when we talk about embedding, it's essentially just saying we've trained the neural network already. Now we're using it to give us a low rank representation of our data. Does that make sense? Yes?

AUDIENCE: I notice that some of the slides have used loss and the others cost. Are they interchangeable?

SARA BEERY: No. This is me being a little fast and loose. The way we defined it is that the loss is like a function over all of the data points. And then the cost might be the overall calculation of that loss over, maybe, all of your data or something.

But I think it's a little semantic. Yeah. It's not the end of the world to think of them somewhat interchangeably. Essentially, both are you're defining some mechanism for evaluating optimality. Yeah?

AUDIENCE: How easy is it to switch between optimizing parameters versus optimizing inputs?

SARA BEERY: So lucky for us, things like PyTorch exist. And so actually this is just a matter of defining what you want to update, basically where you want to freeze your gradients, and where you don't want to freeze your gradients. And that's not actually so hard to do.

AUDIENCE: OK. If we were to do that switch, could we use the prior work, convert it in a way that we can use the prior [INAUDIBLE]?

SARA BEERY: Yeah, yeah. So this is where that modularity comes in. Sorry, guys. If just if you could be quiet. So this is where the modularity comes in, right? So we talked about here, you can have these models where some parts are programmed by a human and some parts are actually being programmed by backprop.

But you could also think of the ones that are quote, unquote programmed by a human. Those might have just actually previously been programmed by backprop. And now you're taking those weights and plugging them in. Yeah?

AUDIENCE: [INAUDIBLE]

SARA BEERY: Yeah.

AUDIENCE: That was through [INAUDIBLE]?

SARA BEERY: Sorry, if everyone could just keep it quiet so that I can hear the questions. Thank you so much. Yeah?

AUDIENCE: [INAUDIBLE] attention block [INAUDIBLE]?

SARA BEERY: Is that what I would call a what?

AUDIENCE: Attention block.

SARA BEERY: An attention block. Oh, so this is a cosine similarity. And cosine similarity is a component of attention, but it doesn't actually make up all of what we would call an attention block. So an attention block generally defined for a transformer or something, will also include the projections into the shared space where you're going to take that similarity, as well as a projection out into the space where you actually want to use the information about what's valid. Yeah?

AUDIENCE: I noticed that [INAUDIBLE].

SARA BEERY: It'll be by the end of the day today, sorry. We were working on it on our meeting right before this. And I thought it might be up, but not quite yet. Yes? Did you have a question? No? OK. Yes?

AUDIENCE: [INAUDIBLE]

SARA BEERY: You're going to need to be a little louder. Sorry. If we can just be quiet as we leave. Thank you so much. All right.

AUDIENCE: So what's the difference between the backdrop for MLP and DAG? Because they are both computational graphs, right?

SARA BEERY: Yeah, so there's no difference. And this is where the only two things you need to take all the stuff we talked about from backprop through MLPs and translate it to backprop through any DAG are these. You need a merging operation and a branching operation. So now, essentially, they're just changed with merging and branching. And that represents any DAG.

AUDIENCE: OK.

SARA BEERY: Yeah Yes?

AUDIENCE: This is just a different way of looking at backpropagation.

SARA BEERY: Yeah, so essentially all we're saying is that all the really detailed examples we showed of backpropagation, those were all for a really simple chain-based DAG. It's just you have a component, and it goes, and a component, and it goes, so it was all a chain.

And now what we're saying is that you can actually take everything we talked about it and apply it to any structure of a DAG. Because now it's just chains, where sometimes they merge, and sometimes they branch. But there's really simple operations that we can use to pass the gradients through merges and branches. And so that's how you expand that definition of backprop to any DAG. Yeah?

AUDIENCE: [INAUDIBLE]

SARA BEERY: Sorry, what?

AUDIENCE: [INAUDIBLE]

SARA BEERY: Sorry. I'm just I'm having trouble hearing you.

AUDIENCE: What is the [INAUDIBLE] for the gradients?

SARA BEERY: Mm-hmm.

AUDIENCE: [INAUDIBLE]

SARA BEERY: Yeah.

AUDIENCE: [INAUDIBLE]. Why is it a sum?

SARA BEERY: Why is it a sum? Because the change in this, as it's coming in from that and that, you're able to just aggregate it. And the reason for that is, essentially, I'm not going to talk this through really simply right now. Come to office hours, and we can talk about it. Yes?

AUDIENCE: So I know that PyTorch builds its computation graph dynamically. For instance, like imagine that in your forward pass you have a conditional, which is like if the magnitude of the gradient exceeds whatever, then I want to apply a CLIP operation. I have a slightly different use case for research.

Do you know how it handles if you apply like a non-torch operation to a tensor? If you apply some of UDF which is just written using normal Python arguments, does it basically just suddenly say in the computation graph, even if I use torch operators before that node in the graph, it is now not optimizable. And I'm just going to treat that basically as the way I would treat a data input?

SARA BEERY: So there might be hacks around this. But generally, if you want to have any operation within a neural network defined in PyTorch, you have to define it as a torch operation, which is basically you are required to define a gradient for that operation. So you have to assume that everything's differentiable. And this is where things get nasty.

Because if something doesn't have a simple gradient-- so there's a lot of built in stuff in PyTorch that handles what those approximations to those gradients would look like. But if you have something that's really complicated, for example, like an occupancy model of species, which is something that we work with, then you actually might have to define that as a torch operation. It also depends.

Sometimes there are choices you can make based on whether or not that makes sense in the structure. You might not actually want to learn all the way through that thing, if it becomes computationally intractable to calculate the gradient.

AUDIENCE: So I guess that's my question is, will PyTorch-- say you didn't define the gradient for that operation or how to compute it, can PyTorch still optimize the rest of the model and just treat what comes in from the output of your UDF as basically being like, you had input some x sub i, x of basically a data vector.

SARA BEERY: I might just define that separately. I might define that as part of your pre-processing for your data. So the pre-processing steps don't necessarily need to be differentiable, things like data augmentation.

So you could put that UDF in your pre-processing, your data loader. And then whatever comes out of it would then be the data that you were going to optimize towards. That's probably the simplest thing to do. But again, computational complexity can get really nasty with some of this stuff, so maybe that's not too bad.

But I've definitely run into issues where I've tried to build simple statistical models into machine learning and with varying levels of success. And often it's just like, OK, it's technically possible, but it's intractable to actually learn. Yeah?

AUDIENCE: On this example, for the superscript a and b, what do they really represent?

SARA BEERY: Oh, it's intended it's intended to be-- it's intended to be open ended. So they could literally just be a duplicate of copy of both. Or it could be some splitting of the embedding vector, or et cetera, et cetera. So it's any branching operation. Yeah. Any differentiable branching operation. OK, one last question.

AUDIENCE: I have a [INAUDIBLE] question.

SARA BEERY: Yeah.

AUDIENCE: The problem set-- was that supposed to be [INAUDIBLE]?

SARA BEERY: It will be there soon. I thought it was up already, but it's not.

AUDIENCE: 3:00 PM.

SARA BEERY: 3:00 PM.

AUDIENCE: OK.

[LAUGHS]

SARA BEERY: All right. Thank you, guys, so much. See you soon.